Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN ENXEÑARÍA DO SOFTWARE

Emulación de dispositivos wearables y monitorización remota inteligente de pacientes pediátricos

Estudiante: Pablo Paz Varela
Dirección: Carlos Fernández Lozano
Marcos Gestal Pose

A Coruña, junio de 2020.

A mis padres

Agradecimientos

Me gustaría expresar mi agradecimiento al Dr. Carlos Fernández y al Dr. Marcos Gestal, directores de este proyecto, por su gran ayuda y excelente trato. Sin su apoyo no habría sido posible la realización de este trabajo.

En especial, agradecer a mis padres el esfuerzo y sacrificio que han realizado para brindarme la posibilidad de estudiar, y por ofrecerme siempre toda la ayuda posible.

También quiero agradecer a mis hermanos, por la confianza depositada en mí y porque han supuesto un apoyo fundamental a lo largo de esta etapa.

Por último, dar las gracias a mis amigos, porque todo es más fácil con sus ánimos.

A todos vosotros, muchas gracias.

Resumen

Las convulsiones febriles afectan del 2 al 5% de los niños entre los seis meses y los 5 años de edad. En la mayoría de casos, se trata de convulsiones benignas que no requieren de ningún tratamiento, lo cual no evita una situación de gran estrés para los padres o cuidadores. Para tratar de mitigar este problema, se ha desarrollado una aplicación que permita monitorizar en tiempo real el estado de un niño durante procesos febriles, alertando ante situaciones que requieran la atención de sus cuidadores. Entre las condiciones de alerta se encuentran el incremento rápido de la temperatura, valores por encima de un umbral ajustable de temperatura y frecuencia cardíaca, orientación boca abajo del paciente y movimientos bruscos repetitivos que puedan indicar la ocurrencia de convulsiones. Esta aplicación se desarrolló para utilizar los dispositivos wearables de MbientLab como fuente de los datos de movimiento, temperatura y frecuencia cardíaca. Estos dispositivos son muy interesantes por la gran flexibilidad que ofrecen, siendo útiles para multitud de propósitos a un precio razonable. Utilizan Bluetooth Low Energy para transmitir los datos y una de sus mayores ventajas es la sencillez de su uso a través de un API que evita en gran medida tener que abordar los detalles de la comunicación. Por estos motivos, también ha formado parte de este trabajo desarrollar una aplicación móvil que pretende emular las funcionalidades básicas de estos dispositivos. El principal objetivo de esta segunda aplicación es cubrir las funcionalidades que utiliza la aplicación de monitorización, para así servir como herramienta para su desarrollo. De esta manera, además de que pueda ser encontrada en un escaneo y poder establecer la conexión correctamente, se pueden enviar datos de temperatura, frecuencia cardíaca, orientación y aceleración. Para producir los datos de movimiento se utiliza el acelerómetro y magnetómetro del teléfono que ejecuta la aplicación, mientras que la temperatura y frecuencia cardíaca se controlan a través de la interfaz.

Abstract

Febrile seizures affect 2 to 5% of children between 6 months and 5 years of age. In most cases, this seizures are benign and do not require any treatment. This does not prevent parents or caregivers from suffering great stress when this situation arises. In order to try to alleviate this problem, a mobile app has been developed that allows to monitor the child condition in real time, alerting if the situation requires of the caregivers atention. Some of the alert conditions are a fast rise in temperature, temperature and heart rate values above a user defined

threshold, the child facing down and repetitive shaking motion that could imply the ocurrence of seizures. This app was designed to use the MbientLab wearable devices as the source of movement, temperature and heart rate data. This devices are interesting because they offer great flexibility, being useful for multiple purposes at a reasonable price. They use Bluetooth Low Energy to transmit data, and one of their advantages is the ease of use through an API that prevents the developer from tackling the communication details. The development of a mobile app that tries to emulate some of the basic capabilities of these devices was also part of this project. The main goal of this second app is to cover the functionality used by the monitoring app, in order to be useful as a development tool. In this manner, other apps using the MbientLab API are able to scan it, successfully establish a connection and receive temperature, heart rate, orientation and acceleration data. To produce the movement data, the accelerometer and magnetometer from the device executing the app are used, while temperature and heart rate are adjusted through user interface elements.

Palabras clave:

- Convulsiones febriles
- Bluetooth Low Energy
- Wearable
- Sensores
- Monitorización
- Alertas
- Android

Keywords:

- Febrile seizures
- Bluetooth Low Energy
- Wearable
- Sensors
- Monitoring
- Alerts
- Android

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1 Motivación

Las convulsiones febriles son la forma más común de convulsiones en la infancia, y afectan a entre el 2 y el 5% de la población pediátrica[1]. La International League Against Epilepsy define las convulsiones febriles como *"los eventos convulsivos asociados a un proceso febril en ausencia de una infección del sistema nervioso central o desequilibrio electrolítico agudo en niños mayores de 1 mes de edad sin convulsiones afebriles previas"*[2] (citado en [1, 3]). Son más comunes entre los 6 meses y los 5 años de edad, con una incidencia máxima alrededor de los 18 meses, y siendo raras a partir de los 7 años.

Normalmente, las convulsiones febriles se producen durante las primeras 24h tras el inicio de la fiebre, y se observan a menudo durante un incremento rápido de la temperatura[1]. Las convulsiones febriles pueden clasificarse en simples o complejas. Una convulsión simple es aislada, breve y generalizada. Por el contrario, una convulsión compleja es focal, múltiple (más de un episodio convulsivo en el mismo proceso febril) o prolongada, durando más de 15 minutos. La mayoría de convulsiones febriles son simples[3].

El riesgo de mortalidad asociado a las convulsiones febriles es extremadamente bajo. Además, los niños con convulsiones febriles simples apenas ven incrementado el riesgo de desarrollar epilepsia respecto a la población general. En el caso de las convulsiones febriles complejas, sí se asocian con un mayor riesgo de epilepsia posterior[1]. En la mayoría de casos no se requiere ningún tipo de tratamiento. Aproximadamente un tercio de los niños con una primera convulsión febril experimentarán una recurrencia, el 10% tendrán tres o más recurrencias[1, 3].

A pesar de tratarse de una forma de convulsiones benigna en la mayoría de casos[1, 3],

causan gran ansiedad e incluso pánico en padres o cuidadores. Esta es la principal motivación para desarrollar una aplicación que permita monitorizar ciertas constantes vitales, alertando ante situaciones que puedan requerir la atención de estos. Una solución de este tipo podría proporcionar cierta tranquilidad a los cuidadores sin necesidad de que estén constantemente junto al niño, lo que redundaría en una mejora de su calidad de vida.

Para la monitorización de ataques epilépticos y distinguir entre estos y aquellos no epilépticos, el estándar es la monitorización de EEG (electroencefalografía) y vídeo[4]. Los electrodos de EEG son incómodos para el paciente, el coste de aplicar este enfoque es elevado y resulta complicado llevarlo a cabo fuera del entorno de un hospital. Por otra parte, múltiples estudios describen el uso de dispositivos wearables para la detección de convulsiones epilépticas utilizando los datos de uno o varios acelerómetros[5–8]. Este enfoque es interesante por su reducido coste y porque permite la monitorización durante periodos prolongados de manera no intrusiva y en un entorno doméstico.

Por estos motivos, para la obtención de los datos se utilizará un dispositivo wearable, que dispondrá de un acelerómetro y sensores de temperatura y frecuencia cardíaca. Se pretende poder detectar en tiempo real los movimientos bruscos que típicamente se producen en las convulsiones febriles, a partir de los datos de aceleración obtenidos. También se pretende alertar en caso de aumentos pronunciados de la temperatura corporal, que podrían anticipar la aparición de convulsiones febriles y ayudar a prevenirlas. Finalmente, según [9], los cambios en la frecuencia cardíaca son una señal clínicamente útil para la detección de crisis convulsivas, por lo que su monitorización también resulta provechosa.

Por otra parte, resulta interesante el desarrollo de un emulador de los dispositivos wearables a utilizar por varias razones. Podemos desarrollar la aplicación de monitorización antes de disponer del dispositivo físico, lo que permite comprobar de antemano las funcionalidades necesarias y minimizar riesgos antes de comprar el hardware final. Proporciona también flexibilidad en el desarrollo al no trabajar con sensores reales, pudiendo manipular los datos enviados.

Además, la oferta de dispositivos wearables ha crecido mucho en los últimos años, pero la mayoría de ellos son productos cerrados que difícilmente se pueden utilizar para desarrollar este tipo de soluciones. Los dispositivos wearables de MbientLab son muy interesantes por la flexibilidad y facilidad de desarrollo que ofrecen, siendo útiles para multitud de aplicaciones. Por lo tanto, disponer de un emulador que pueda ser fácilmente extensible podría ser útil para otros desarrolladores. Finalmente, el desarrollo de este emulador permitirá aumentar el conocimiento sobre el funcionamiento de este tipo de dispositivos y las tecnologías que utilizan, como Bluetooth Low Energy.

1.2 Objetivos

El fin de este proyecto es el desarrollo de dos aplicaciones móviles: una aplicación que emule las capacidades básicas de los dispositivos wearables de MbientLab y una aplicación que permita la monitorización de pacientes pediátricos con procesos febriles.

Los objetivos principales para la aplicación de emulación:

- Capacidad de realizar *advertising* para poder ser escaneada y permitir la conexión de aplicaciones que utilicen el API proporcionado por MbientLab.
- Implementar el comportamiento necesario para que el API pueda recibir datos de temperatura, frecuencia cardíaca, aceleración y orientación; permitiendo que el usuario pueda controlar los datos producidos.
- Diseño modular que facilite añadir nuevos modelos de dispositivo, así como la implementación de más sensores.

Los objetivos principales para la aplicación de monitorización de pacientes pediátricos:

- Escaneo y conexión a dispositivo wearable de MbientLab.
- Obtención de datos en tiempo real de sensores del dispositivo.
- Detección de situaciones de riesgo, alertando al usuario en consecuencia.
- Configuración de ciertos parámetros de las alertas.
- Mostrar gráficamente los datos recibidos en tiempo real
- Mostrar gráficamente valores máximos y mínimos diarios durante la última semana.
- Funcionamiento en segundo plano

Los modelos de wearables que ofrece actualmente MbientLab no se adaptan completamente al caso de uso planteado, por lo que el objetivo es desarrollar una aplicación a modo de prueba de concepto. En caso de que se desease desarrollar un wearable para este fin basado en la tecnología de MbientLab, se pretende que la aplicación ControlFlu pueda funcionar con este, con modificaciones mínimas en caso necesario.

1.3 Estructura de la memoria

Este documento contine los siguientes capítulos:

- **Capítulo 2 - Fundamentos tecnológicos:** Se describen brevemente las tecnologías utilizadas y los motivos de su elección.
- **Capítulo 3 - Estado de la cuestión:** Se muestran las alternativas encontradas en el mercado y se valora por qué es útil este proyecto.
- **Capítulo 4 - Planificación y evaluación de costes:** Se describe la planificación y seguimiento del proyecto, y se recogen los costes de su realización.
- **Capítulo 5 - Desarrollo:** Se describe la metodología utilizada y se detalla en profundidad el proceso seguido para la realización del proyecto.
- **Capítulo 6 - Resultados:** Se valora el resultado obtenido y la consecución de los objetivos planteados.
- **Capítulo 7 - Conclusiones:** Se exponen las conclusiones obtenidas fruto de la realización de este trabajo.
- **Capítulo 8 - Futuros desarrollos:** Se enumeran funcionalidades que no forman parte de los objetivos de este proyecto, pero podrían ser interesantes para ampliarlo en el futuro.

Capítulo 2

Fundamentos tecnológicos

2.1 Aplicaciones en dispositivos móviles

2.1.1 Android

Android e iOS son los dos sistemas operativos móviles más populares. Tanto es así que entre los dos copan prácticamente el 100% de la cuota de mercado, según Gartner[10] (Ver tabla 2.1).

Android es un sistema operativo móvil basado en Linux y desarrollado por Google. Con una cuota de mercado global de un 88%, es el sistema para móviles más popular en la actualidad. Se trata de un proyecto de código abierto destinado a funcionar en una gran cantidad de dispositivos, con factores de forma muy diferentes. Esto se contrapone con el modelo cerrado que emplea Apple con iOS, el cual está diseñado y optimizado exclusivamente para sus teléfonos iPhone.

El origen de Android se encuentra en la Open Handset Alliance, un consorcio de compañías del sector tecnológico y móvil, lideradas por Google, que se unieron para definir están-

Sistema Operativo	Unidades 2Q18	Cuota de mercado 2Q18	Unidades 2Q17	Cuota de mercado 2Q17
Android	329,503.4	88.0 %	321,848.2	87.8 %
iOS	44,715.1	11.9 %	44,314.8	12.1 %
Otros SO	112.1	0.0 %	433.1	0.1 %
Total	374,330.6	100.0 %	366,596.1	100.0 %

Tabla 2.1: Cuota de mercado de SO móviles. Fuente: Gartner (Agosto de 2018)

dares abiertos y realizar un esfuerzo común en el desarrollo de una plataforma abierta para dispositivos móviles.

Normalmente las aplicaciones para Android se desarrollan en el lenguaje de programación Java, pero también está soportado oficialmente el lenguaje Kotlin. También se puede desarrollar parte de la aplicación en C o C++ utilizando las herramientas proporcionadas por el kit de desarrollo nativo(NDK), útil para lograr latencias bajas o en operaciones computacionalmente intensivas; así como la utilización de bibliotecas escritas en estos lenguajes.

Arquitectura de la plataforma

Android es una pila de software de código abierto. Los componentes principales de la plataforma son los siguientes:

- **Kernel de Linux:** Es la base de la plataforma, proporcionando al ART funcionalidades subyacentes como generación de subprocesos y la administración de memoria de bajo nivel. Además, Android aprovecha sus funciones de seguridad y permite a los fabricantes de dispositivos desarrollar controladores de hardware para un kernel conocido.
- **Capa de abstracción de hardware(HAL):** Se encarga de brindar interfaces estándares que exponen las capacidades de los componentes hardware del dispositivo al Framework.
- **Android Runtime(ART):** Es el entorno de ejecución para las aplicaciones desde Android 5.0, reemplazando al entorno de ejecución Dalvik. Las mejoras introducidas por ART incluyen la compilación ahead-of-time(AOT), recolección de basura optimizada, conversión de los archivos Dalvik Executable format(DEX) a un código máquina más compacto a partir de Android 9 y un mejor soporte para la depuración.
- **Bibliotecas nativas en C/C++:** Muchos de los componentes y servicios fundamentales de Android están construidos con código nativo que requiere bibliotecas nativas escritas en C y C++. La plataforma permite a las aplicaciones acceder a las funcionalidades de algunas de estas bibliotecas a través de APIs del Framework de Java, como por ejemplo el API Java de OpenGL para dibujar y manipular gráficos.
- **Java API Framework:** Todas las funcionalidades disponibles en el sistema Android se ponen a disposición de los desarrolladores a través de APIs escritas en Java. Los desarrolladores tienen acceso completo a las mismas APIs del Framework que utilizan las aplicaciones del sistema.

- **Aplicaciones del sistema:** Android trae una serie de aplicaciones predeterminadas para los contactos, mensajería, correo electrónico, calendario y navegación por internet, entre otras. Éstas no poseen un estatus especial, y pueden ser sustituidas por aplicaciones externas como aplicación predeterminada para estas funciones.

Componentes fundamentales de una aplicación Android

- **Activity:** se trata del componente principal sobre el que se construyen las aplicaciones de Android. Proporciona la ventana sobre la que se dibuja la interfaz de usuario y, normalmente, cada activity corresponde a una pantalla de la aplicación. Cuando se inicia una aplicación, Android crea la activity correspondiente e invoca el código definido en sus callbacks. La práctica habitual es definir la interfaz de usuario de la pantalla en un archivo *xml*, que se utiliza en el momento de crear la activity. Una vez inicializados, la activity gestiona las interacciones del usuario con los elementos que conforman la interfaz. Una aplicación suele estar formada por múltiples activities, que trabajan conjuntamente para proporcionar una experiencia coherente al usuario. A menudo, cada una de ellas se centra en una tarea concreta o funcionalidad. Las activities tienen normalmente una dependencia mínima entre ellas, e incluso pueden ser invocadas por aplicaciones externas si así se desea. Un ejemplo de esto podría ser una activity para escoger un contacto, para escoger una fotografía o para redactar un nuevo email. Un aspecto fundamental de este componente, que también se aplica a otros como fragment o service, es que está diseñado para definir su comportamiento según un ciclo de vida. El ciclo de vida está formado por un conjunto de estados en los que se puede encontrar una actividad desde su creación hasta que el sistema la destruye (ver figura 2.1). Una activity va pasando por los diferentes estados en función de la navegación del usuario y de la situación del sistema. Cuando la activity cambia de estado, el sistema ejecuta los callbacks correspondientes, que permiten ajustar el comportamiento a la nueva situación. Por ejemplo, si es necesario gastar recursos en la actividad, se recomienda obtenerlos en el momento en que esta pase a un estado visible y liberarlos cuando el usuario ya no la va a ver. Cuando una acitivity pasa a estar inactiva, el sistema puede destruirla si escasean los recursos. También se destruye cuando se produce algún cambio de configuración, como por ejemplo cuando el usuario rota la pantalla del dispositivo. Además, una vez destruida, el sistema recreará la activity en su estado inicial, por lo que se debe guardar el estado de esta si se desea que el usuario se encuentre la pantalla igual que antes. El desarrollador debe tener en cuenta estos aspectos y gestionar los cambios de estado del componente cuidadosamente, un paradigma que también se aplica a otros componentes utilizados para la construcción de aplicaciones en Android.

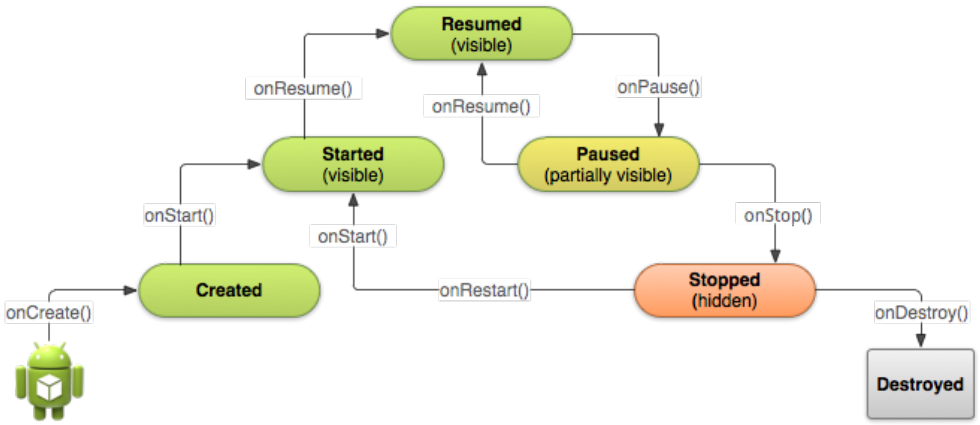


Figura 2.1: Ciclo de vida de una actividad

- **Fragment:** encapsulan una porción de la interfaz de usuario y su comportamiento. Funcionan como secciones modulares de una activity, que tienen su propio ciclo de vida dependiente del de la activity a la que pertenezcan, y que se pueden añadir o eliminar dinámicamente de estas. Se puede, por ejemplo, combinar varios fragments en una misma pantalla para formar una interfaz multipanel que puede adaptar la disposición de los paneles según el tamaño del dispositivo. También se puede utilizar el mismo fragment en múltiples pantallas diferentes. Permite aplicar un diseño modular a partes de la interfaz, lo que favorece la reutilización de código y dar un aspecto coherente en la aplicación a elementos similares. Este enfoque facilita la creación de interfaces de usuario dinámicas y flexibles.

- **Service:** es un componente que puede realizar operaciones de larga duración en segundo plano y que no ofrece interfaz de usuario. Una vez iniciado puede continuar ejecutándose aunque el usuario cambie de aplicación. Otros componentes pueden iniciar un service o enlazarse a él. Cuando son iniciados, se ejecutan indefinidamente hasta que algún componente los detenga o se detengan a sí mismos. Cuando otros componentes se enlazan, el service se inicia automáticamente y se destruye cuando no quede ningún componente enlazado. Ambos mecanismos se pueden combinar para conseguir un service que se ejecute indefinidamente y que, al mismo tiempo, los componentes que lo necesiten se enlacen para interactuar con él. Otra distinción que hace Android es entre services en segundo plano y primer plano. Los primeros realizan operaciones que no percibe directamente el usuario mientras que los services en primer plano ejecutan operaciones que sí puede notar. Android obliga a que los services en primer plano muestren una notificación permanentemente, y les otorga mayor prioridad. Esta prioridad es importante ya que en situaciones en las que el sistema tenga pocos recursos,

habrá pocas posibilidades de que Android destruya un servicio ejecutándose en primer plano.

- **Intent**: es un objeto utilizado para la comunicación entre componentes de la aplicación que describe de forma abstracta una operación a realizar. Se utiliza para iniciar una activity, para iniciar o enlazarse a un service y también pueden enviarse en forma de broadcasts que recibirán otros componentes interesados. El contenido de un intent consiste principalmente en una acción y los datos incluidos. Los intents pueden ser de dos tipos: explícitos o implícitos. Los intents explícitos definen la activity o service de destino sin ambigüedad, y suelen utilizarse dentro de la misma aplicación para desencadenar la navegación entre activities o la utilización de services. En el caso de los intents implícitos, a partir de la acción a realizar y el tipo de datos incluidos, el sistema busca una activity de destino apropiada, lo cual se utiliza para iniciar activities externas a la aplicación. Por ejemplo, si se utiliza un intent con la acción estándar para visualizar datos, incluyendo una ubicación geográfica, se solicita que otra aplicación la muestre en un mapa. En caso de esperar un resultado de la activity externa, este se devuelve en otro intent. Finalmente, los componentes de la aplicación pueden registrar uno o varios broadcast receivers, a través de los cuales reciben comunicaciones de eventos a través de broadcast intents con las acciones que les interesen. Este mecanismo no está limitado al ámbito de la aplicación. Pueden enviarse entre aplicaciones y también lo utiliza el sistema para notificar algunos eventos, como cambios en alguna configuración del sistema.

2.1.2 Android Jetpack

Jetpack es un conjunto de bibliotecas, herramientas y guías desarrolladas por Google que facilitan la implementación de funcionalidades comunes a multitud de aplicaciones, ayudando a desarrollar aplicaciones de calidad que sigan prácticas modernas de diseño[11]. El hecho de estar separadas del Framework de la plataforma permite que se actualicen más a menudo que éste y que puedan ofrecer retrocompatibilidad. A continuación se describen brevemente algunas de estas bibliotecas que han sido utilizados para la realización de este trabajo:

- **Bibliotecas de soporte:** ayudan a desarrollar aplicaciones para múltiples versiones del sistema, proporcionando algunas de las nuevas funcionalidades en versiones anteriores. En otros casos, presenta alguna alternativa de funcionalidad similar que se adecúe a la versión en la que se esté ejecutando. Su principal ventaja radica en evitar en la mayoría de casos tener que implementar funcionalidades con varias versiones del API, teniendo que comprobar en tiempo de ejecución la implementación adecuada.

- **Notification**: Componentes que permiten crear y gestionar notificaciones. Las notificaciones son mensajes que se muestran fuera de la interfaz de la aplicación, útiles para presentar alertas, recordatorios, comunicaciones o cualquier información oportuna de la aplicación.
- **Preference:** biblioteca que facilita la creación de pantallas de configuración, encargándose de manejar tanto el almacenamiento de los ajustes como la interfaz de usuario, proporcionando una experiencia consistente en todos los dispositivos y versiones del sistema.
- **ViewModel:** es una clase diseñada para almacenar y administrar datos relacionados con la interfaz de una manera optimizada para los ciclos de vida . Es una práctica recomendada utilizar ViewModel para evitar que los controladores de IU tengan la responsabilidad de obtener datos, con la ventaja adicional de que los datos sobreviven a cambios de configuración de la interfaz, como rotaciones de pantalla.
- **Livedata:** es una clase de retención de datos observable que, a diferencia de un observable normal, está optimizado teniendo en cuenta el ciclo de vida de sus observadores. De esta forma, no notificará a actividades, fragments o servicios que no se encuentren activos.
- **ROOM:** es una biblioteca de persistencia que proporciona una capa de abstracción para SQLite, facilitando su uso sin dejar de aprovechar toda su potencia.

2.1.3 Herramientas

Android Studio

Android Studio es el IDE (Entorno de Desarrollo Integrado) oficial para el desarrollo de aplicaciones para Android. Está basado en IntelliJ Idea, por lo que cuenta con su potente editor, debugger, completado de código inteligente, herramientas de análisis y refactorización de código, integración del control de versiones y multitud de herramientas de desarrollo. También cuenta con un buen ecosistema de plugins.

Adicionalmente, Android Studio ofrece otras funcionalidades con el objetivo de aumentar la productividad en el desarrollo de aplicaciones entre las que se encuentran:

- Emulador, que permite una ejecución más rápida que en un dispositivo real.
- Herramienta para perfilar el rendimiento de la aplicación.

- Herramienta para diseñar interfaces de usuario.
- Integración con Gradle.
- Aplicar cambios a la aplicación sin tener que reiniciarla.
- Herramienta Lint que advierte de posibles problemas de compatibilidad de versiones, de usabilidad o de rendimiento.

Gradle

Gradle es una herramienta de código abierto para la automatización de la construcción de software centrado en el rendimiento y la flexibilidad. Entre sus ventajas respecto a otras herramientas destacan que está diseñado para ser altamente personalizable y extensible; es rápido al reutilizar todo lo posible resultados de ejecuciones anteriores y ejecutar tareas en paralelo; y soporta muchos lenguajes y tecnologías populares.

Además, es la herramienta de compilación oficial de Android, estando incluida en forma de plugin en el sdk y disponiendo de muy buena integración en Android Studio.

Git

Git es un sistema de control de versiones distribuído diseñado para mejorar la velocidad y eficiencia en el desarrollo. Se publica bajo una licencia libre y es el sistema más utilizado en la actualidad por delante de su principal competidor, SVN.

Al ser un sistema distribuido, la mayoría de operaciones se realizan localmente, lo que incrementa mucho la velocidad en los flujos de trabajo habituales. Otra de sus características principales es su sistema de ramas, que fomenta experimentar en diferentes líneas de desarrollo de forma aislada, antes de decidir si queremos integrarlas en el desarrollo principal. Frente a otros sistemas, las operaciones con las ramas son muy rápidas en Git. Los sistemas centralizados tienen la ventaja de ser más sencillos de aprender. Sin embargo, la curva de aprendizaje de Git se compensa con la mayor flexibilidad y control que ofrece.

MetaWear y MetaBase

Además de las pequeñas aplicaciones en forma de tutorial que se centran en cómo utilizar algún módulo o característica concreta de los dispositivos, se encontraron 2 aplicaciones en la tienda de aplicaciones Google Play que MbientLab proporciona a modo de ejemplo y

que sirven para familiarizarse con los sensores y funciones disponibles. Ambas aplicaciones pueden escanear, conectarse, configurar y recibir datos de los sensores disponibles. También permiten comprobar si hay alguna actualización de firmware e instalarla en el dispositivo a través de BLE.

La aplicación MetaBase[12] permite recibir datos tanto en *streaming* como descargando los datos guardados en la memoria del periférico. Sin embargo, al menos en su versión para Android, no permite visualizarlos en la propia app. Permite exportar los archivos generados en la sesión a servicios como Google Drive o Dropbox, enviarlos a través de correo electrónico o sincronizarlos con el servicio MetaCloud de MbientLab.

Para el desarrollo de este proyecto se ha utilizado en mayor medida la aplicación llamada MetaWear, debido a que permite visualizar los datos recibidos en tiempo real dentro de la propia aplicación. Esta se utilizó tanto como toma de contacto con el dispositivo MetaTracker, sirviendo para descubrir sus capacidades y familiarizarnos, como para posteriormente verificar que nuestro emulador conseguía establecer la conexión y enviar correctamente los datos de aceleración. En el momento de la redacción de esta memoria, esta aplicación ya no se encuentra disponible en Google Play, y en la página de MbientLab únicamente referencian la aplicación MetaBase para comprobar el funcionamiento de sus dispositivos. El código fuente todavía puede obtenerse en GitHub[13].

nRF Connect for Mobile

Esta aplicación para Android, desarrollada por Nordic Semiconductor, es una herramienta muy útil para el desarrollo de dispositivos que utilicen BLE. Es capaz de escanear, conectarse y realizar las operaciones definidas en el estándar. Una funcionalidad especialmente interesante de esta herramienta es la de definir pruebas automatizadas a través de un fichero *xml*, haciendo uso de las operaciones y capacidades implementadas en la aplicación.

Al inicio del desarrollo resultó de utilidad para comprobar el contenido de los paquetes de *advertising* y la estructura de servicios y características del dispositivo MetaTracker analizado. Posteriormente, también se utilizó esta herramienta para implementar una suite de pruebas automatizada.

2.2 Bluetooth Low Energy

2.2.1 Descripción general de Bluetooth

La tecnología Bluetooth es un sistema de comunicaciones de corto alcance pensado para sustituir los cables en la conexión de dispositivos electrónicos. Entre sus principales características se encuentran la robustez, el bajo consumo de energía y bajo precio[14, p. 180]. Hay dos sistemas definidos de comunicación Bluetooth: Basic Rate (BR) y Low Energy(LE). Ambos sistemas operan en la banda de frecuencia ISM [1] de 2.4 GHz y ambos proporcionan mecanismos para el descubrimiento de dispositivos, establecimiento de la conexión y mantenimiento de la conexión. El SIG (Special Interest Group) de Bluetooth está formado por más de 34000 compañías y se encarga de desarrollar las especificaciones de la tecnología Bluetooth.

Bluetooth BR/EDR

También conocido como el Bluetooth clásico, se trata del estándar que ha ido evolucionando con la especificación de Bluetooth desde su versión 1.0 . El sistema Basic Rate incluye las extensiones opcionales EDR(Enhanced Data Rate) y AMP(Alternate MAC/PHY). La tasa de transmisión de datos es de 721.2 Kb/s para el modo Basic Rate, y de un máximo de 2.1 Mb/s con Enhanced Data Rate. Utilizando un controlador secundario AMP pueden alcanzarse tasas de bits de hasta 54 Mb/s a través de un enlace IEEE 802.11. Soporta múltiples niveles de potencia, de 1mW a 100 mW, y una topología de red punto a punto que está optimizada para el streaming de audio.

Bluetooth LE

Bluetooth Low Energy no es una optimización del Bluetooth clásico, sino una tecnología nueva con unos objetivos completamente distintos, y ambos estándares son incompatibles entre sí. El sistema LE está diseñado para permitir su uso en dispositivos que requieren un menor consumo energético, menor complejidad y menor coste que el de BR/EDR. Soporta tasas de bits desde 125 Kb/s hasta un máximo de 2 Mb/s y múltiples niveles de potencia, desde 1mW hasta 100 mW[15]. También permite múltiples topologías de red: punto a punto para transmisión de datos, broadcast para servicios de localización y red en malla para creación de redes de dispositivos a gran escala.

[1] Industrial, Scientific and Medical: Bandas de frecuencia reservadas internacionalmente para usos industriales, científicos o médicos y para las que no se necesita licencia.

Bluetooth Low Energy se incluyó por primera vez en la especificación de Bluetooth 4.0 en el año 2010. Por su bajo consumo de energía, ésta tecnología es ideal para la comunicación inalámbrica con dispositivos wearables, siempre y cuando las capacidades de rango y velocidad de transmisión de datos sean adecuadas para el caso de uso. BLE no pretende abarcar otras necesidades de comunicación inalámbrica, siendo más adecuadas en tal caso otras tecnologías como el Bluetooth clásico, WiFi o NFC.

2.2.2 Algunos conceptos de BLE

A continuación definimos algunos términos fundamentales relativos a BLE:

- **Attribute Protocol(ATT):** Este protocolo define dos roles, rol de servidor y rol de cliente. Permite que un servidor exponga un conjunto de atributos accesibles por un cliente[14]. Un atributo es un valor discreto y se identifica su tipo con un identificador único universal(UUID). El cliente puede descubrir, leer y escribir atributos. El servidor dispone de mecanismos para notificar o indicar, lo que proporciona una manera eficiente de enviar valores de atributos al cliente sin que éste inicie la lectura.
- **Generic Attribute profile(GATT):** Este perfil define el formato de servicios y características utilizando el protocolo ATT. También define la manera de emplear ATT para descubrir, leer, escribir u obtener indicaciones de estos atributos. Todos los perfiles existentes se basan en GATT, aunque en el futuro podrían aparecer perfiles que no hagan uso de GATT.
- **Servicio:** Un servicio es una colección de características, usualmente asociadas a un tipo de datos o funcionalidad.
- **Característica:** Una característica contiene un valor, y opcionalmente puede contener múltiples descriptores.
- **Descriptor:** Un descriptor es un atributo definido que describe el valor de una característica.

Podemos distinguir los roles que toman los dispositivos en la comunicación BLE siguiendo dos criterios. A la hora de establecer la conexión, un dispositivo ejerce el rol de periférico y otro el de central. El periférico realiza el *advertising*, y el central escanea en busca de dispositivos que estén en proceso de *advertising*. Una vez establecida la conexión, en función de cómo se comuniquen, los dispositivos pueden tener el rol de servidor GATT o de cliente GATT. Ambos dispositivos pueden funcionar como servidor o cliente para diferentes servicios y características.

2.3 Dispositivos biométricos/wearables

Se han considerado varios dispositivos wearables como plataforma para la recogida de los datos biométricos y su envío en tiempo real a la aplicación de monitorización que nos disponemos a desarrollar. A continuación se describen las características principales de los tres más interesantes.

2.3.1 Dispositivos MetaWear de MbientLab

(a) MetaTracker (MTR) (b) MetaMotion C (MMC) (c) MetaMotion R (MMR)

Figura 2.2: Modelos de sensores disponibles en la web de MbientLab

MbientLab fabrica varios dispositivos wearables que permiten la recogida de datos de movimiento y múltiples sensores a través de Bluetooth Low Energy. Estos dispositivos son ideales para el desarrollo de wearables para fitness, investigación deportiva y reconocimiento de gestos.

Actualmente MbientLab ofrece tres modelos de estos dispositivos(ver figura 2.2). Los tres modelos incluyen acelerómetro, giroscopio, sensor de temperatura, sensor de presión barométrica y sensor de luz ambiental. También poseen un botón físico y un LED rgb. El modelo MetaTracker es el único que no tiene magnetómetro, pero se añade un sensor de humedad que no incluyen los otros dos (ver tabla 2.2). Además, todas las placas incluyen pines GPIO y bus I2C que permiten añadir sensores externos. Para nuestra aplicación de monitorización sería necesario añadir un sensor PPG para la frecuencia cardíaca. La precisión de este tipo de sensores podría ser suficiente incluso aunque se integrase con el wearable en una pulsera[16, 17]. Adicionalmente, en caso de que se encontrase que el sensor de temperatura incluido no resulte adecuado para medir la temperatura corporal, sería necesario añadir uno externo. MbientLab también ofrece la posibilidad de producir dispositivos, firmware y software personalizados si se lo requiere.

Sensor	Mag	Hum	Fusion	Memoria	Energía	SoC	Precio
MTR		X		4MB	pila de botón	nRF51822	$60.99
MMC	X		X	8MB	pila de botón	nRF52832	$75.99
MMR	X		X	8MB	batería lipo recargable	nRF52832	$79.99

Tabla 2.2: Diferencias entre los sensores

Toda la comunicación con los dispositivos se realiza utilizando BLE, y los datos producidos por los sensores pueden enviarse en tiempo real o guardarse en la memoria flash del dispositivo para ser descargados posteriormente. Además, puede realizarse cierto procesamiento sobre los datos antes de ser enviados o guardados. Entre las operaciones soportadas se incluyen numerosos modos de filtrado y operaciones matemáticas. Las operaciones pueden encadenarse, e incluso pueden programarse comandos en respuesta a alguna condición en los datos (un ejemplo sería encender el led cuando un valor supera un determinado umbral).

Una de las mayores ventajas que ofrecen estos dispositivos es el API de código abierto que su fabricante pone a disposición en forma de biblioteca, el cual facilita mucho el desarrollo de aplicaciones que haga uso de estos dispositivos. El API se encarga de manejar todos los aspectos de la comunicación BLE.

En cuanto a la seguridad, estos dispositivos utilizan el SoC nRF51822 con el SoftDevice S110 de Nordic Semiconductor, el cual soporta al completo la especificación de seguridad de la versión 4.1 de Bluetooth Low Energy. Sin embargo, ninguna de las placas ofertadas ni el API proporcionado soportan ninguna de estas características de seguridad. Cabe mencionar que cuando se realiza la conexión con una de estas placas, ésta deja de hacer *advertising*, con lo cual no se podría conectar nadie más. Los parámetros de *advertising* pueden configurarse y en conjunción con el uso de macros (serie de instrucciones que se pueden escribir en la placa para que se ejecuten cada vez que arranca) podría limitarse que extraños puedan conectarse a ella. Por ejemplo, haciendo que la placa solo sea visible un tiempo determinado tras pulsar el botón físico de la placa. A pesar de todo, existen dispositivos capaces de hacer *sniffing* del tráfico Bluetooth en el aire por lo que los datos transmitidos, al no ir cifrados, podrían ser interceptados.

MbientLab ha lanzado varias campañas en Kickstarter en el pasado, para financiar el desarrollo de algunos de los modelos de placas que han comercializado. Especialmente interesante, por su idoneidad para el caso de uso que nos atañe, es una campaña que han lanzado para un producto llamado MetaHealth[18]. Este estaría enfocado a aplicaciones de salud y fitness, y prometía integrar sensores de frecuencia cardíaca, temperatura y sudoración, además del acelerómetro y giroscopio comunes al resto de modelos. Sin embargo, hace mucho que se debería de haber fabricado y todo parece indicar que MbientLab ha abandonado el desarrollo de este

modelo.

Ante la expectativa de que en un futuro próximo fabriquen algún modelo orientado específicamente a la monitorización de la salud, se ha decidido desarrollar una aplicación móvil para Android que emule las funciones básicas necesarias para el desarrollo de nuestra aplicación ControlFlu y otros desarrollos futuros.

2.3.2 Movisens EcgMove 4

Figura 2.3: Movisens EcgMove 4 – ECG and Activity Sensor

El Movisens EcgMove 4 es un sensor orientado a la utilización por parte de investigadores que quieran obtener datos de ECG y actividad de alta calidad. Se coloca en el pecho utilizando dos electrodos adhesivos o una cinta, y su carcasa es resistente al agua. Contiene sensores de ECG de un canal, aceleración, giro, presión atmosférica y temperatura. No se pueden añadir sensores externos adicionales. Los datos sin procesar se almacenan en el sensor para ser descargados posteriormente a través de usb. El dispositivo cuenta con una memoria interna de 4GB, con la que el fabricante especifica un máximo de dos semanas de grabación de datos. Su batería LiPo dura aproximadamente tres días con el Bluetooth apagado[19].

También puede analizar ciertos parámetros en el propio sensor y enviar los resultados en tiempo real a través de Bluetooth LE. Cada parámetro analizado produce un valor por minuto y no se pueden transmitir los datos brutos a través de Bluetooth. No dispone de un API que se encargue de la comunicación a través de BLE, aunque sí que se proporciona una biblioteca que simplifica esta tarea. Esta biblioteca proporciona los UUIDs de los servicios y características que pueden leerse y una forma conveniente de interpretar los datos recibidos.

Cabe mencionar que Movisens también tiene otros modelos de sensores, como el EdaMove 4, capaz de medir la actividad electrodérmica, o el LightMove 4 que dispone de un sensor de

luz ambiental. Estos modelos no tienen sensor de ECG pero sí comparten el resto de sensores y características del EcgMove 4.

2.3.3 Maxim MAXREFDES101#: health sensor platform 2.0

Figura 2.4: Maxim MAXREFDES101#: health sensor platform 2.0

Plataforma que aglutina varios productos de maxim en el factor de forma de un reloj y que permite la captura de varias bioseñales importantes para la salud. Es un producto orientado a evaluar los diferentes sensores y componentes para facilitar el desarrollo de dispositivos wearables personalizados utilizando las tecnologías de Maxim Integrated. Se puede personalizar el comportamiento de la plataforma utilizando el entorno de desarrollo ARM mbed para desarrollar un firmware propio. La plataforma se compone por:

- Placa micro que incluye un microcontrolador, un circuito integrado de gestión de energía, Bluetooth modo dual y acelerómetro y giroscopio de 6 ejes.
- Placa sensor que incluye un sensor de frecuencia cardíaca (y oxímetro) óptico, un sensor de ECG, un sensor de temperatura corporal, un acelerómetro de 3 ejes y un concentrador de sensores biométricos que tiene un algoritmo embebido para la obtención de la frecuencia cardíaca.
- Placa Pico adaptor que se utiliza para actualizar el firmware de la placa micro.
- Carcasa en forma de reloj, batería y cables necesarios.

La conexión a la placa puede realizarse a través de usb o a través de Bluetooth Low Energy. Maxim integrated proporciona el firmware necesario y aplicaciones para el control y lectura de los sensores tanto para Windows como para Android. El precio por unidad es de 399$[20].

2.4 Elección de las tecnologías

Se ha escogido como plataforma de sensorización las placas wearables fabricadas por MbientLab por tratarse del producto que mejor se adapta a nuestras necesidades. Permite la integración de todos los sensores que se pretenden utilizar para la monitorización de las convulsiones febriles, es la más adecuada para la transmisión de los datos en tiempo real utilizando BLE y en relación a otras opciones su precio es muy razonable. Además, MbientLab proporciona APIs para múltiples sistemas operativos y lenguajes de programación que facilitan la comunicación con el dispositivo wearable, lo cual supone una ventaja importante en el desarrollo de aplicaciones que hagan uso de sensores.

A la hora de decidir el sistema operativo móvil para el cuál desarrollar ambas aplicaciones se han considerado las siguientes cuestiones:

- La cuota de mercado de los dispositivos Android es mucho mayor que la de iOS en todo el mundo, y en España es de hasta el 90%.
- Para desarrollar aplicaciones para iOS hay que disponer de un iPhone y de un Mac, ya que el entorno de desarrollo xcode y las herramientas de desarrollo nativo sólo funcionan en macOS. En cuanto a Android, el entorno de desarrollo oficial, Android Studio, puede utilizarse en Windows, Mac y Linux. Tanto xcode como Android Studio son gratuitos. En general, el desarrollo para iOS supone un mayor coste económico.
- El precio de los iPhone puede ser una barrera para los potenciales usuarios de nuestra aplicación. En el caso de Android la oferta de dispositivos es muy amplia, abarcando todo el rango de precios y prestaciones. Por otra parte, los usuarios de iPhone están más dispuestos a pagar por las aplicaciones que los de Android.
- No se disponía de un iPhone ni un Mac, y sí se podía disponer de dos terminales Android.

También se valoró la posibilidad de desarrollar la aplicación de monitorización para ambos sistemas utilizando el framework Apache Cordova. Esta opción no está soportada oficialmente por MbientLab, pero se ha encontrado un plugin de Cordova para el API de MetaWear[21]. Sin embargo este plugin no se ha actualizado en más de 3 años, habiendo cambiado el API de

versión más recientemente. Tampoco soporta el uso de todas las funcionalidades y sensores del API. Por tanto, se ha desechado esta posibilidad.

Teniendo en cuenta todas las consideraciones anteriores la opción preferida es Android. Una vez comprobado que la realización de la aplicación de emulación era viable en el sistema Android y considerando que el desarrollo para ambos sistemas operativos de forma nativa sería un proyecto demasiado grande para este trabajo, se ha optado definitivamente por el desarrollo para Android de ambas aplicaciones.

Se ha escogido utilizar el IDE Android Studio. Éste es el entorno de desarrollo oficial para Android, y como tal, cuenta con el soporte de Google. Está basado en el potente IDE IntelliJ IDEA y facilita la integración con multitud de herramientas y componentes de desarrollo del ecosistema. Además, es la plataforma soportada por MbientLab y tanto los tutoriales como la documentación hacen referencia a este IDE.

En cuanto al lenguaje de programación, se ha decidido utilizar Java por contar con alguna experiencia anterior con este. Se ha escogido Gradle como herramienta de automatización de la compilación por ser la herramienta oficial para Android y por contar con un gran soporte en Android Studio. Asimismo, se ha utilizado ROOM y SQLite para la persistencia al ser la opción sugerida en la documentación de Android. Para el control de versiones se ha optado por utilzar Git por ser la opción más utilizada en la actualidad y el servicio proporcionado por Gitlab por ofrecer repositorios privados y de forma gratuita.

Capítulo 3

Estado de la cuestión

En este capítulo se exponen aquellos productos encontrados que son de interés por su similitud con las aplicaciones a desarrollar, y se discute el valor ofrecido por este proyecto frente a las alternativas disponibles actualmente.

3.1 Aplicaciones que emulan periféricos BLE

3.1.1 BLE Peripheral Simulator

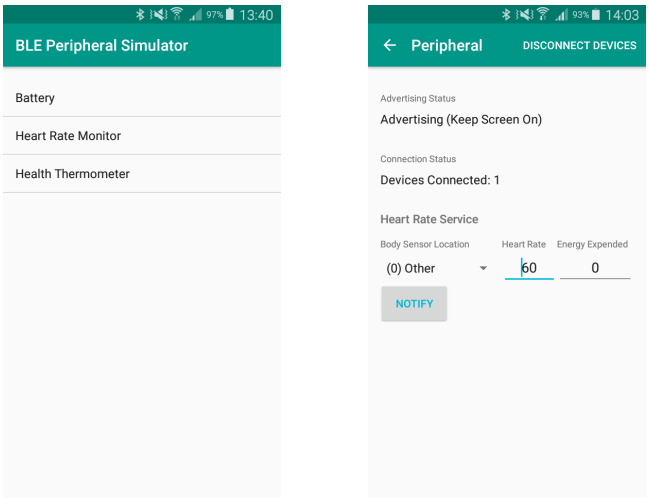


Figura 3.1: BLE Peripheral Simulator

BLE Peripheral Simulator[22] es una aplicación para Android capaz de simular un periférico BLE con uno de entre tres servicios de la especificación estándar: Servicio de Batería,

Servicio de Frecuencia Cardíaca o Servicio de Temperatura. Desde la aplicación puede ajustarse el valor de la característica. Otros dispositivos pueden conectarse a esta para leer y escribir la característica, así como subscribirse a notificaciones para recibir el valor cuando cambie.

Al no permitir definir servicios y características personalizados, no podemos replicar la estructura del servidor GATT de los dispositivos MetaWear utilizando esta aplicación. El código fuente de esta aplicación está disponible en *Github*, lo que resulta interesante como ejemplo de la utilización de las APIs de BLE de Android.

3.1.2 nRF Connect for Mobile

Esta aplicación[23] es una herramienta genérica muy completa, que permite explorar dispositivos BLE y comunicarse con ellos. Permite escanear los dispositivos BLE cercanos, parsear los datos del advertisement, conectarse a dispositivos que sean conectables, descubrir servicios y características parseando aquellos adoptados por el SIG, leer o escribir características y habilitar o deshabilitar notificaciones e indicaciones.

En cuanto al rol de periférico, esta app es capaz de realizar *advertising* BLE en dispositivos que lo soporten, y permite configurar un servidor GATT personalizado al que otros dispositivos pueden conectarse y operar sobre sus características. De esta forma, se puede replicar la jerarquía de servicios del servidor GATT de un dispositivo MetaWear. Sin embargo, no se puede definir ningún comportamiento en función de los comandos que se escriban en una característica, por lo que no se pudo conseguir que el API de MetaWear establezca la conexión con esta app.

3.1.3 Conclusiones

Las aplicaciones aquí descritas y otras muy similares permiten utilizar un dispositivo Android compatible como periférico BLE, crear servicios y características GATT, y permitir la lectura de valores definidos por el usuario para estas características. Esta funcionalidad es útil para el desarrollo de aplicaciones que vayan a realizar operaciones sobre características cuyo único fin es almacenar una medición o valor, como ocurre al comunicarse con periféricos que utilicen servicios adoptados por el SIG de Bluetooth. Sin embargo, los wearables de MbientLab utilizan un servicio y dos características personalizados para recibir comandos y responder al dispositivo central. Las apps existentes no permiten simular dicho comportamiento.

Por este motivo, se consideró que los desarrolladores de aplicaciones para estos dispositivos posiblemente demanden una herramienta que facilite el proceso de desarrollo. Además, también se deseaba poder controlar en tiempo real los datos de los sensores a enviar, y

transmitir los datos de aceleración producidos por el acelerómetro que incluyen la inmensa mayoría de teléfonos inteligentes. Estas funciones tampoco se implementan en ninguna de las aplicaciones encontradas.

3.2 Monitorización de pacientes pediátricos

3.2.1 Pulseras de actividad

(a) Fitbit Charge 4 (b) Apple Watch Series 5

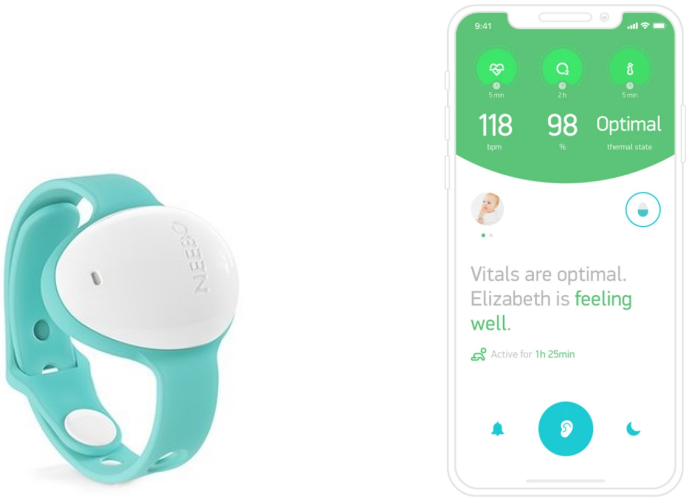
Figura 3.2: Monitores de actividad

En los últimos años, han salido al mercado infinidad de dispositivos wearables destinados a la monitorización del ejercicio físico, en forma de pulseras y smartwaches. Fabricantes como Fitbit, Samsung, Apple, Garmin, Xiaomi y muchos otros, han apostado por lanzar sus propuestas de monitores de actividad para llevar en la muñeca.

Habitualmente, este tipo de dispositivos disponen de acelerómetro, giroscopio, magnetómetro, sensor de frecuencia cardíaca y GPS, entre otros. Cada dispositivo tiene su aplicación móvil con la que se sincronizan y a través de la cual proporcionan principalmente datos de gasto energético, pasos, distancia recorrida, nivel de actividad física y patrones del sueño. En cuanto al precio, se pueden encontrar dispositivos de este tipo en un amplio rango de precios. Como referencia, el precio de uno de los dispositivos más populares, el Fitbit Charge 4, es de 149,95 €.

El gran crecimiento de la oferta de pulseras de actividad y smartwaches, puede ofrecer la oportunidad de aprovechar los datos que generan este tipo de dispositivos para aplicaciones en el ámbito de la salud y la investigación, siempre teniendo en cuenta algunas limitaciones en la precisión de sus datos[24–26].

3.2.2 Neebo



(a) Pulsera Neebo (b) Aplicación para iOS

Figura 3.3: Solución de monitorización Neebo

Neebo[27] es un dispositivo wearable en forma de pulsera para niños de 0 a 5 años. Dispone de sensores con los que monitoriza la frecuencia cardíaca, saturación de oxígeno, estado térmico y actividad del niño. Además, incluye un sistema de alertas que avisa en caso de que los signos vitales estén fuera del rango definido o se produzcan cambios en el nivel de actividad. A partir de la actividad, también registra si el niño está durmiendo. También se muestran algunas estadísticas, como las horas de sueño.

La pulsera utiliza BLE para transmitir los datos, y puede conectarse directamente al teléfono del cuidador o a la base de carga. Los datos pueden sincronizarse con un servicio en la nube a través del teléfono o la base de carga, que dispone de conectividad Wi-Fi. A través de este servicio en la nube, otros cuidadores pueden recibir los datos sin estar conectados a la pulsera directamente.

La aplicación necesaria para utilizar este dispositivo sólo está disponible para iOS y el coste de la pulsera con la base de carga es de 299 dólares.

3.2.3 Conclusiones

Como ya se ha mencionado, al examinar los productos wearables disponibles comercialmente, se ha encontrado una oferta muy amplia en cuanto a smartwaches y pulseras de fitness. Sin embargo, a pesar de este crecimiento en el mercado de los wearables, la búsqueda de alternativas pone de manifiesto que apenas existen productos enfocados a monitorizar a pacientes

pediátricos en sus casas. Casi la totalidad de los dispositivos disponibles están enfocados en la monitorización de la actividad física y el seguimiento de ciertas estadísticas relevantes al rendimiento deportivo del usuario.

Por lo tanto, se considera que existe una posible necesidad para la que actualmente apenas existen productos disponibles. Frente a las alternativas encontradas, el producto a desarrollar pretende ofrecer una solución económica, disponible en el sistema Android y que proporcione una experiencia sencilla centrada en la monitorización de niños con fiebre o episodios convulsivos.

Capítulo 4

Planificación y evaluación de costes

4.1 Planificación y seguimiento

El inicio de este proyecto se planificó para el día 2 de octubre de 2019. Una vez definidos los objetivos fundamentales para cada aplicación, se dividió el trabajo a realizar en iteraciones de un tamaño estimado similar. Se planificó una duración aproximada de cinco semanas para cada iteración. Cada iteración incluye las fases de análisis, diseño, implementación y pruebas.

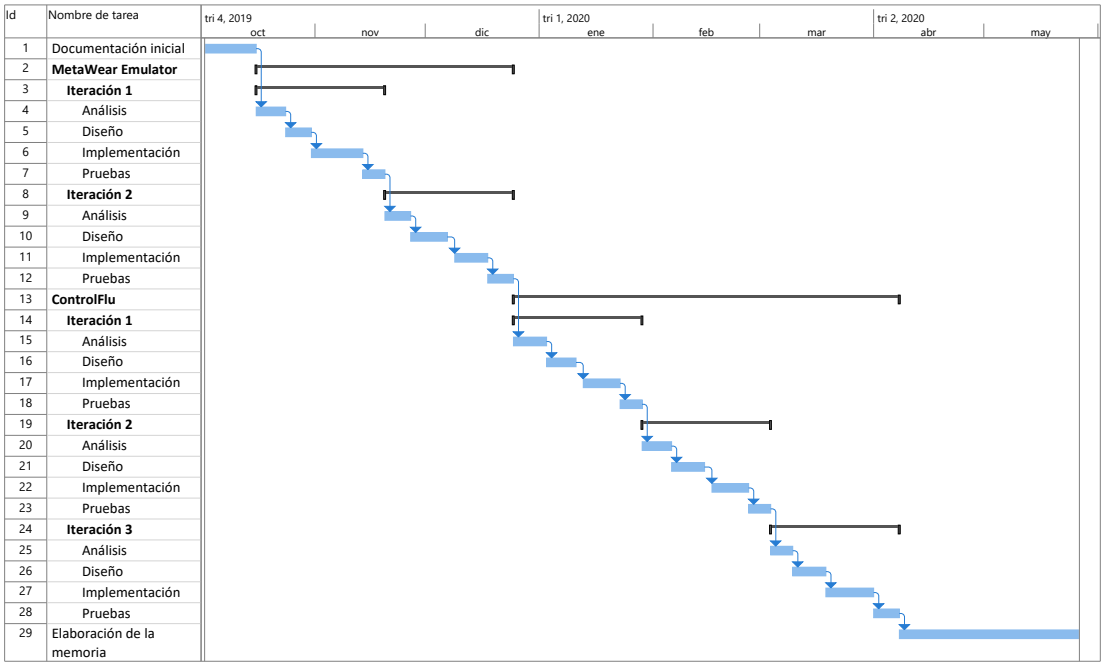


Figura 4.1: Diagrama de Gantt

Id	Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
1	Documentación inicial	10 días	mié 02/10/19	mar 15/10/19	
2	**MetaWear Emulator**	**50 días**	**mié 16/10/19**	**mar 24/12/19**	
3	**Iteración 1**	**25 días**	**mié 16/10/19**	**mar 19/11/19**	
4	Análisis	6 días	mié 16/10/19	mié 23/10/19	1
5	Diseño	5 días	jue 24/10/19	mié 30/10/19	4
6	Implementación	10 días	jue 31/10/19	mié 13/11/19	5
7	Pruebas	4 días	jue 14/11/19	mar 19/11/19	6
8	**Iteración 2**	**25 días**	**mié 20/11/19**	**mar 24/12/19**	
9	Análisis	5 días	mié 20/11/19	mar 26/11/19	7
10	Diseño	8 días	mié 27/11/19	vie 06/12/19	9
11	Implementación	7 días	lun 09/12/19	mar 17/12/19	10
12	Pruebas	5 días	mié 18/12/19	mar 24/12/19	11
13	**ControlFlu**	**75 días**	**mié 25/12/19**	**mar 07/04/20**	
14	**Iteración 1**	**25 días**	**mié 25/12/19**	**mar 28/01/20**	
15	Análisis	7 días	mié 25/12/19	jue 02/01/20	12
16	Diseño	6 días	vie 03/01/20	vie 10/01/20	15
17	Implementación	8 días	lun 13/01/20	mié 22/01/20	16
18	Pruebas	4 días	jue 23/01/20	mar 28/01/20	17
19	**Iteración 2**	**25 días**	**mié 29/01/20**	**mar 03/03/20**	
20	Análisis	6 días	mié 29/01/20	mié 05/02/20	18
21	Diseño	7 días	jue 06/02/20	vie 14/02/20	20
22	Implementación	8 días	lun 17/02/20	mié 26/02/20	21
23	Pruebas	4 días	jue 27/02/20	mar 03/03/20	22
24	**Iteración 3**	**25 días**	**mié 04/03/20**	**mar 07/04/20**	
25	Análisis	4 días	mié 04/03/20	lun 09/03/20	23
26	Diseño	7 días	mar 10/03/20	mié 18/03/20	25
27	Implementación	9 días	jue 19/03/20	mar 31/03/20	26
28	Pruebas	5 días	mié 01/04/20	mar 07/04/20	27
29	Elaboración de la memoria	35 días	mié 08/04/20	mar 26/05/20	28

Tabla 4.1: Planificación inicial

Como se puede observar en el diagrama de Gantt de la figura 4.1, las tareas se planificaron secuencialmente al disponer de un solo desarrollador en cada momento. Según la planificación inicial, se estimó como fecha de finalización de este proyecto el día 26 de mayo de 2020.

Durante el seguimiento de la planificación a lo largo del proyecto se pudo comprobar que se produjeron desviaciones importantes sobre el tiempo estimado inicialmente, lo cual se refleja en el diagrama de la figura 4.2. En la primera iteración de la aplicación de emulación, se encontraron problemas para lograr el establecimiento de la conexión. Se requirió de más tiempo del esperado para el estudio del funcionamiento del API de MbientLab, lo que se tradujo en un retraso considerable en el proyecto. Además, una vez implementada esta funcionalidad, un fallo que causaba inestabilidad en la conexión ralentizó el desarrollo durante el resto de la iteración y el inicio de la segunda, cuando se pudo encontrar una solución.

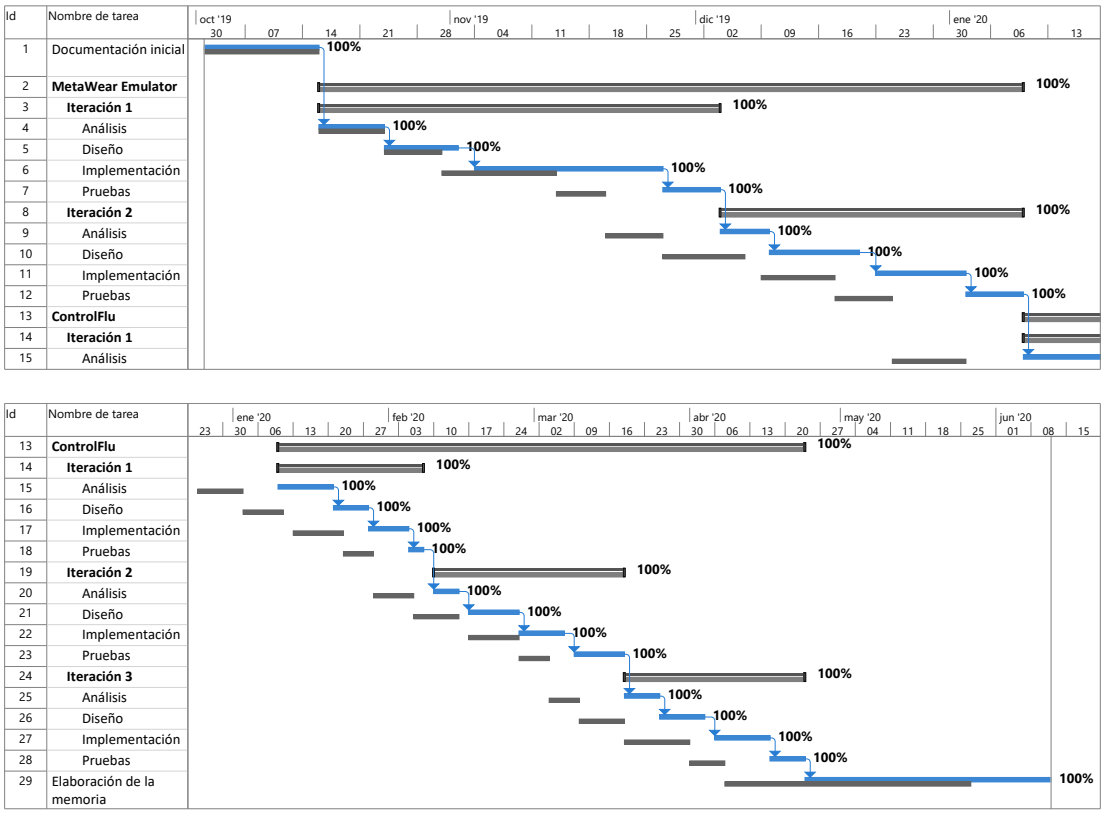


Figura 4.2: Diagrama de Gantt de seguimiento

También cabe mencionar que se produjo otro pequeño contratiempo en el mes de marzo como consecuencia de la situación generada por la pandemia de COVID-19. Los cambios para adecuarse a esta situación se tradujeron en unos días de inactividad durante la fase de pruebas de la segunda iteración de la aplicación ControlFLu. Esto no impidió cumplir con el plazo que se había estimado para esta aplicación, gracias en buena parte a que se pudo terminar la primera iteración con 4 días de antelación. Esto fue gracias a la utilización de una biblioteca para realizar el escaneo de los dispositivos y a que alguna funcionalidad guardaba ciertas similitudes con las implementadas en las iteraciones anteriores, como por ejemplo la implementación de un servicio en segundo plano. El resto de tareas se realizaron en un tiempo similar al estimado.

La fecha de finalización real fue el día 11 de junio de 2020, lo que significó un retraso de más de dos semanas respecto a la fecha planificada. Se puede concluir que la planificación inicial fue un poco optimista, teniendo en cuenta la falta de experiencia con las tecnologías y el riesgo de alguna de las tareas a realizar. Durante la realización de la segunda aplicación este factor tuvo menor influencia, por la experiencia adquirida en las iteraciones previas.

4.2 Evaluación de costes

En cuanto a los costes materiales, se incluyen en la tabla 4.2. El dispositivo Metatracker de la empresa MbientLab se utilizó para estudiar el funcionamiento de los wearables de este fabricante y su comunicación a través de BLE. Para el desarrollo fue necesario disponer de un ordenador personal adecuado para utilizar el entorno de desarrollo y las herramientas necesarias. No hubo ningún coste relativo al uso de software. También hubo que disponer de dos teléfonos móviles Android que soportasen la comunicación a través de BLE. Además, al menos uno de ellos debía soportar esta comunicación en el rol de periférico, teniendo la capacidad de realizar *advertising*.

Recurso material	**Precio**	**Cantidad**	**Coste**
PC para el desarrollo	700 €	1	700 €
Wearable MetaTracker	67,09 €	1	55,45 €
Teléfono móvil Android con BLE	200 €	2	400 €
		TOTAL	**1155,45 €**

Tabla 4.2: Coste de recursos materiales

En cuanto a los recursos humanos, se empleó para este proyecto un desarrollador junior, un analista y un jefe de proyecto. Los roles de desarrollador y analista fueron desempeñados por el alumno, mientras que tanto el rol de cliente como el de jefe de proyecto fueron llevados a cabo por los directores del TFG. Estos aportaron el feedback necesario y supervisaron el desarrollo mediante reuniones en cada iteración. Suponiendo la contratación de un desarrollador recién graduado, se estimó su coste a partir del salario mínimo para un desarrollador junior. Según las tablas salariales publicadas en el convenio aplicable al sector[1], este se encuentra en 15.860,56 € anuales. Para el analista y el jefe de proyecto, se estimaron unos salarios anuales de 30000 € y 50000 €, respectivamente. Se puede ver el coste calculado en la tabla 4.3.

Recurso humano	**Precio**	**Horas trabajadas**	**Coste**
Desarrollador junior	8,81 €/hora	1152 horas	10149,12 €
Analista	16,67 €/hora	280 horas	4667,60 €
Jefe de Proyecto	27,78 €/hora	20 horas	555,60 €
		TOTAL	**15372,32 €**

Tabla 4.3: Coste de recursos humanos

[1]Resolución de 22 de febrero de 2018, de la Dirección General de Empleo, por la que se registra y publica el XVII Convenio colectivo estatal de empresas de consultoría y estudios de mercado y de la opinión pública. «BOE» núm. 57, de 6 de marzo de 2018, páginas 26951 a 26981

Capítulo 5

Desarrollo

5.1 Metodología

Para la realización de este proyecto se decidió seguir un modelo de desarrollo iterativo y creciente. Este se caracteriza por tener un ciclo de vida compuesto por múltiples iteraciones consecutivas; y en cada una de ellas no solo se refina el sistema, sino que se le añaden nuevas funcionalidades[28]. Cada iteración debe producir un producto ejecutable y probado. Esta organización en iteraciones permite recibir valoraciones del cliente una vez termina cada iteración, lo que puede permitir una detección temprana de problemas graves de diseño y hacer más manejables posibles cambios de los requisitos. Antes de cada iteración se deciden las funcionalidades a implementar en función de su riesgo y de la prioridad que les da el cliente. Para cada iteración se llevan a cabo las fases de análisis, diseño, implementación y prueba.

Al contar con un sólo desarrollador y tratándose de un proyecto no crítico, se consideró adecuado adoptar un enfoque ágil. Los principios de desarrollo ágiles se centran en priorizar

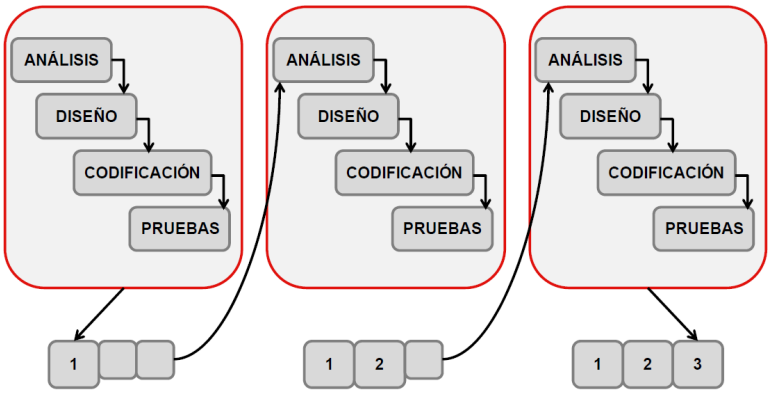


Figura 5.1: Ciclo de vida iterativo

una colaboración muy estrecha con el cliente, no producir más documentación de la necesaria para llevar a cabo el trabajo y adaptarse a los cambios en lugar de seguir un plan de forma estricta. Se busca de esta manera detectar los cambios en los requisitos lo más pronto posible, para intentar mitigar su impacto. En contraposición con enfoques más rígidos o formales, el cliente también tiene más poder para modificar el rumbo del desarrollo.

Se ha escogido esta forma de trabajo porque se trata de una metodología moderna, ágil, flexible, popular, y que ayuda a reducir riesgos y a corregir errores de forma temprana, antes de que se conviertan en demasiado complejos de abordar. Estas características hacen que se adapte perfectamente al trabajo a realizar.

El proceso de desarrollo de este proyecto se divide en dos partes claramente diferenciadas, el desarrollo de una aplicación para Android que simule el dispositivo wearable de sensorización y el desarrollo de otra aplicación para Android que se encargue de monitorizar el estado de un paciente pediátrico en base a los datos recogidos por este tipo de sensores.

Para la primera de ellas, se dividió su desarrollo en dos iteraciones. La primera iteración se enfocó en los aspectos básicos de conectividad BLE mientras que la segunda iteración se ocupó de implementar todo lo necesario para enviar los datos de sensores, controlados por el usuario.

Para la aplicación ControlFlu, el desarrollo se organizó en tres iteraciones. En la primera, se implementó la conectividad y obtención de datos a través del API del fabricante de los dispositivos wearables. La segunda iteración se centró en la funcionalidad relativa a las alertas. Finalmente, en la tercera iteración se implementaron las gráficas en tiempo real, el ajuste de algunos parámetros de las alertas y una pantalla donde el usuario puede ver los valores máximos y mínimos diarios de la última semana, además de realizar los retoques finales del aspecto de la aplicación.

5.2 Documentación inicial

Como fase previa al inicio del desarrollo propiamente dicho, se realizó una toma de contacto con el funcionamiento de los dispositivos fabricados por MbientLab. Para este fin, se pudo disponer de acceso a un dispositivo MetaTracker durante las fases iniciales de este proyecto. Además de utilizar las aplicaciones que el fabricante tiene disponibles para probar las funcionalidades de sus dispositivos y familiarizarnos con su uso, se siguió el tutorial FreeFall[29], que guía al desarrollador en la implementación de una aplicación sencilla que obtiene datos del acelerómetro y los procesa para detectar cuando la placa se encuentra en caída libre.

La configuración en Android Studio del proyecto y las bibliotecas de MbientLab, así como parte del código referente a la conexión con el dispositivo wearable y obtención de datos implementado en la realización del tutorial han servido como punto de partida para el desarrollo de la aplicación ControlFlu.

También se procedió a estudiar la documentación disponible del API de MetaWear, así como la documentación proporcionada por Android para iniciarse en el desarrollo de aplicaciones para este sistema operativo.

5.3 MetaWear Emulator

5.3.1 Iteración 1

En esta primera iteración, el objetivo principal es implementar las funcionalidades relativas al *advertising*, así como conseguir que otro dispositivo que utilice el API de MbientLab pueda establecer una conexión con nuestra aplicación.

Análisis

El objetivo de esta iteración es el desarrollo de una primera versión de la aplicación que pueda ser escaneada por otros dispositivos y que consiga establecer una conexión iniciada por el API de MetaWear.

Requisitos funcionales:

- Comprobaciones bluetooth.
- Iniciar y detener *advertising*.
- Permitir la conexión de la API de MetaWear.
- Mostrar en pantalla el estado del *advertising* y de la conexión.
- Capacidad de devolver el valor de batería del sistema.

Requisitos no funcionales:

- Creación de un servicio en segundo plano que permita mantener la conexión y continuar mandando datos aunque el teléfono esté bloqueado.

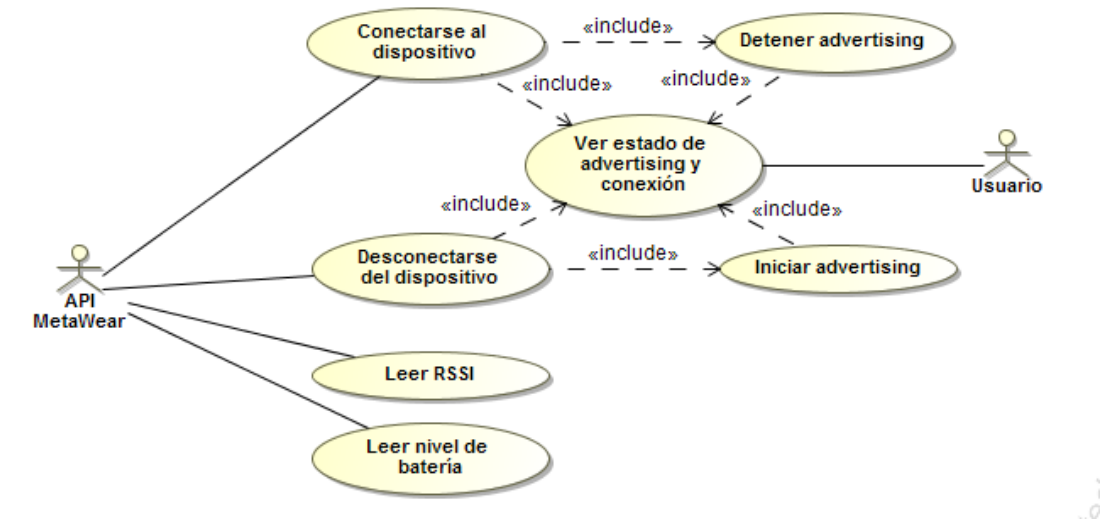


Figura 5.2: Casos de uso de la primera iteración

- Soportar la versión de Android más antigua que permita la utilización del rol de periférico de BLE, para así soportar el mayor número de dispositivos posible.

Los casos de uso detectados para la primera iteración pueden verse en la figura 5.2 (página 34). En esta primera versión, el usuario no interaccionará con la interfaz de usuario. Al iniciarse la aplicación, se iniciará el servicio en segundo plano y comenzará automáticamente a realizar *advertising*. El usuario podrá ver en pantalla si se está realizando *advertising* y si hay algún dispositivo conectado o no.

El actor API MetaWear representa a otro dispositivo que utiliza esta biblioteca para gestionar la conexión BLE y para enviar comandos a nuestro dispositivo. Para imitar lo mejor posible el comportamiento de las placas MetaWear; cuando se establezca una conexión, nuestra aplicación deberá dejar de enviar paquetes de *advertising* y volver a hacerlo una vez se termine la conexión. En la práctica, esto implica que solo se pueda conectar un dispositivo a la vez a nuestra app, como ocurre con los wearables de MbientLab.

Diseño

En el diagrama de clases de la figura 5.3 (página 35) se muestra el diseño final para esta iteración. Los diagramas de clases mostrados a lo largo de esta memoria no representan una especificación completa, para mejorar su legibilidad. De esta manera, se han omitido por ejemplo los métodos del ciclo de vida de Android que comúnmente implementan todas las actividades y servicios (`onCreate()`, `onStop()`, `onStartCommand()`, etc.) y todas las

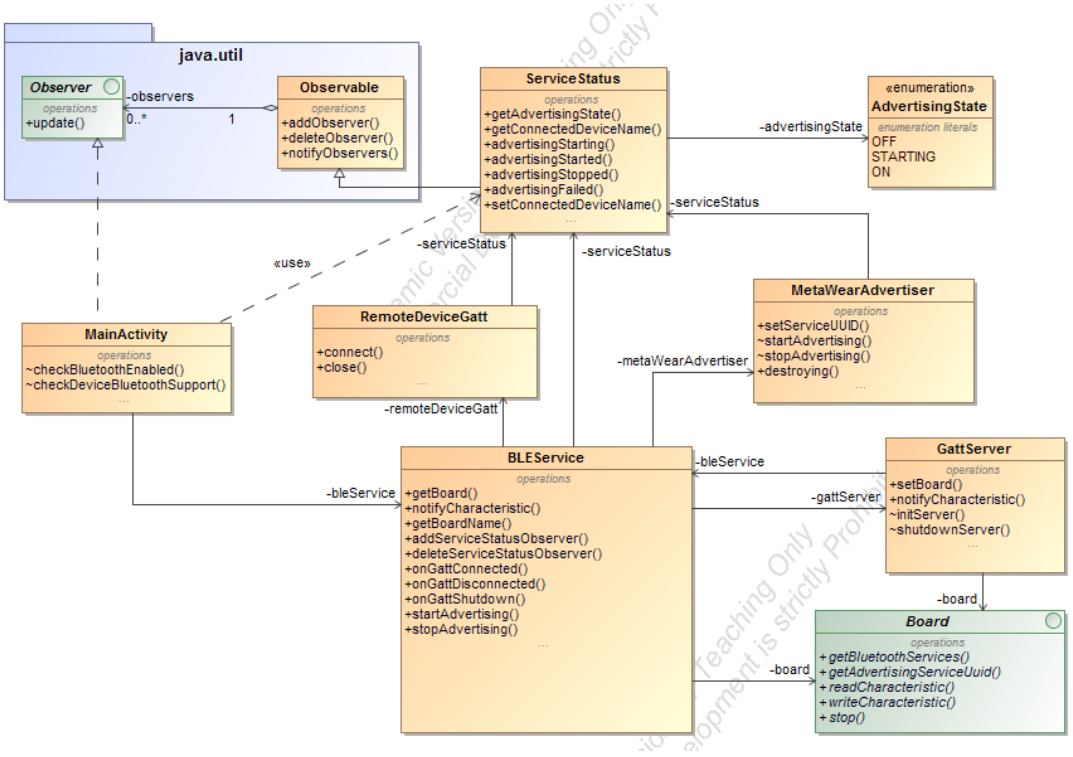


Figura 5.3: Diagrama de clases de la primera iteración

relaciones de dependencia a excepción de aquellas que se consideraron útiles o interesantes.

La clase `BLEService` extiende la clase `Service` de Android, implementando los métodos de ciclo de vida para gestionar un servicio en segundo plano. Esta clase se encarga de crear las instancias y coordina el funcionamiento de las clases `GattServer`, `MetaWearAdvertiser` y `RemoteDeviceGatt`. También se encarga de obtener la instancia de la clase `Board` y pasársela a `GattServer` antes de iniciarlo, así como especificar el UUID de servicio en `MetaWearAdvertiser`. En esta iteración se utilizó únicamente una clase para encapsular el comportamiento de la placa, pero en la siguiente iteración introducimos una jerarquía de clases que permita la elección del modelo de placa deseado.

La clase `GattServer` se encarga de realizar todas las operaciones sobre el servidor GATT y manejar las peticiones de los clientes GATT. Se intentó evitar cualquier comportamiento específico de los dispositivos MetaWear en este componente, delegando cuando fuese necesario en la clase `Board`. De esta manera, la definición de los servicios, características y descriptores se produce en la clase `Board`, que proporciona el método `getBluetoothServices()` para que `GattServer` los obtenga en el momento de añadirlos al servidor. Además, cada vez que un cliente solicita leer o escribir una característica, se llama a los métodos `readCharacteristic()` y `writeCharacteristic()` de la clase `Board`,

respectivamente.

Un objetivo marcado para esta iteración era el de mostrar en pantalla el nombre del dispositivo que establezca una conexión con nuestra app. Sin embargo, en el momento de la implementación, no se encontró manera de obtener dicho nombre por tener el rol de servidor GATT en la conexión. La solución encontrada fue la de abrir una conexión con el rol de cliente, lo que se implementó a través de la clase `RemoteDeviceGatt`. El diagrama de clases mostrado (ver figura 5.3, página 35) refleja este cambio en el diseño.

Para mostrar en la actividad principal el estado de *advertising* y si hay algún dispositivo conectado, se utilizó el patrón observador, aprovechando la clase `Observable` y el interfaz `Observer` del paquete `java.util`. La actividad se enlaza con el servicio en segundo plano y añade su implementación anónima del interfaz `Observer` para recibir las notificaciones de `ServiceStatus`. `BLEService` mantiene una instancia de `ServiceStatus` que pasa a través de sus constructores a `MetaWearAdvertiser` y `RemoteDeviceGatt` para que mantengan actualizado el estado en todo momento.

Implementación

En primer lugar, se implementaron las comprobaciones necesarias relativas al uso de Bluetooth en nuestra aplicación. Al iniciarse la aplicación, se comprueba si el usuario tiene encendido el Bluetooth, y de no ser así, se le solicita que lo encienda. Esto se consigue iniciando una actividad con la acción de intent ACTION_REQUEST_ENABLE, lo que hace que el sistema muestre un cuadro de diálogo para habilitar el Bluetooth. Si el usuario acepta, el sistema habilita las funciones de Bluetooth y el foco vuelve a nuestra aplicación. A continuación, se comprueba que el dispositivo soporte BLE con la llamada `hasSystemFeature(FEATURE_BLUETOOTH_LE)` del `PackageManager`. También se especifica en el manifest que la aplicación requiere de BLE, para que no aparezca disponible en la Play Store para dispositivos que no dispongan de esta característica.

Android 4.3 introdujo soporte para el rol central de BLE, pero el rol de periférico no fue soportado hasta la versión 5.0[30]. Por este motivo, nuestra aplicación no soportará versiones inferiores a Android Lollipop(5.0). También hay que tener en cuenta que no todos los dispositivos que soportan BLE tienen la capacidad de realizar *advertising*, independientemente de la versión del sistema que utilicen. Esto es imprescindible para desempeñar el rol de periférico y, en consecuencia, necesario para el funcionamiento de esta aplicación. Android proporciona el método `isMultipleAdvertisementSupported()` para comprobar si el dispositivo dispone de esta capacidad.

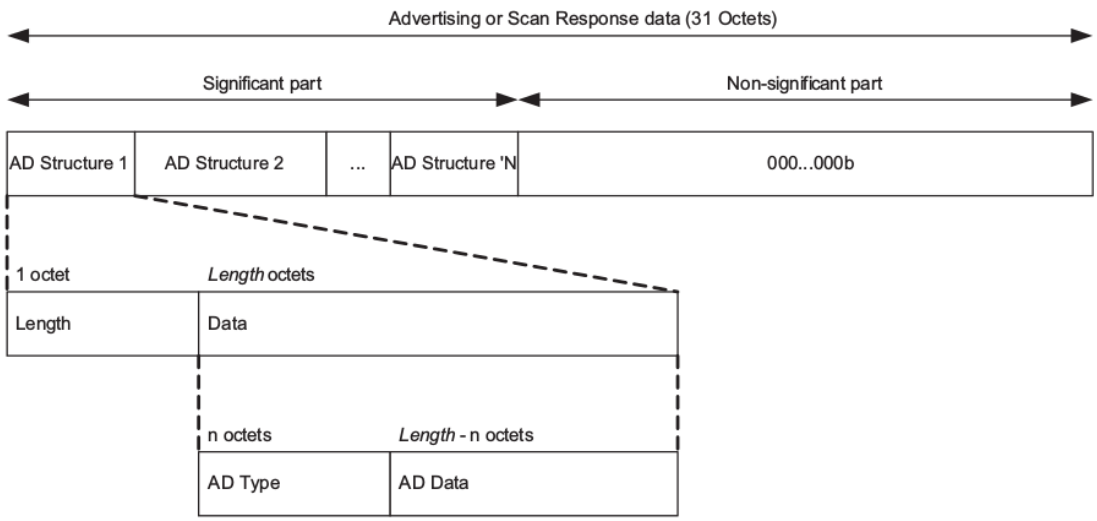


Figura 5.4: Formato de los datos de *advertising* y *scan response*

Si el dispositivo no es compatible con estas funciones o el usuario declinó habilitar el Bluetooth en el diálogo del sistema, la aplicación se cierra mostrando un mensaje al usuario. Cabe mencionar, que se probaron múltiples dispositivos con el fin de comprobar si podían realizar *advertising*, y sólo se pudo disponer de uno que soportase esta función, un Samsung Galaxy A3, que fue el utilizado para el desarrollo de esta aplicación.

El siguiente paso fue el de replicar el proceso de *advertising* que realizan las placas MetaWear. Durante esta iteración se pudo disponer de un dispositivo MetaTracker para su estudio. Resultó muy útil el uso de la aplicación nRF Connect, que es capaz de escanear y mostrar el contenido de los paquetes de *advertising* (ver figura 5.6a, página 39). Las placas incluyen en su *advertising* un UUID propio, no definido por el SIG de Bluetooth, que identifica el servicio utilizado por el API de MetaWear para enviar comandos y recibir notificaciones. Este UUID puede utilizarse para filtrar los resultados a la hora de escanear en busca de estas placas, para que no aparezcan otros dispositivos cercanos. Las aplicaciones de ejemplo de MbientLab realizan este filtrado, por lo que si no incluimos este UUID nuestro dispositivo no será encontrado por estas. Android nos permite añadir UUIDs de servicios a los datos de *advertising* con el método `addServiceUuid()`.

El tamaño máximo total del paquete de *advertising* que contruyamos es de 31 bytes, teniendo en cuenta que para cada dato que incluyamos consumiremos al menos 2 bytes adicionales para indicar su longitud y tipo. En nuestro caso, a los 3 bytes que ocupan los *flags* le sumamos los 18 bytes que ocupa el UUID del servicio, con lo que quedarían 10 bytes libres. Esto significa que no podríamos incluir en el *advertising* un nombre de dispositivo que ocupe más de 8 bytes. Adicionalmente, el dispositivo remoto que realice un escaneo puede solicitar más infor-

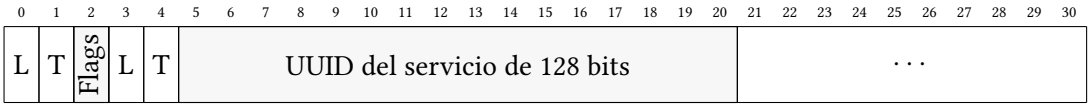
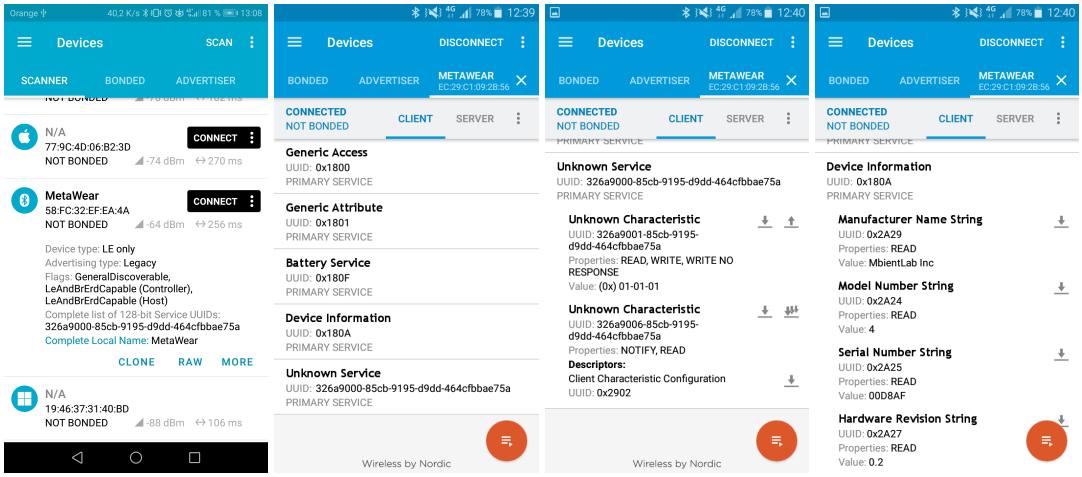


Figura 5.5: Datos de *advertising* que envía nuestra aplicación.

mación sin conectarse enviando un *scan request*, a lo que el periférico responde con un *scan response*. Utilizando un *scan response* se pueden enviar otros 31 bytes de datos, siguiendo la misma estructura que en el paquete de *advertising*. Para poder enviar nombres de dispositivo más largos, estos se incluyen en el *scan response*.

Cabe mencionar que en las pruebas realizadas con el Samsung Galaxy A3 utilizado para el desarrollo, al detener el *advertising* y volver a iniciarlo, la dirección incluida en el *advertising* cambia de forma aleatoria. Ésta es una característica de privacidad recogida en la especificación de Bluetooth cuyo objetivo es dificultar el rastreo de la localización del dispositivo a lo largo del tiempo. En lugar de incluir su dirección real en los paquetes de *advertising*, el dispositivo genera una dirección privada cada cierto tiempo (la especificación recomienda una duración de 15 minutos)[31]. El fabricante puede decidir si utilizar este modo de funcionamiento en sus dispositivos, y en el caso de las placas MetaWear, éstas no cambian su dirección[32]. Sin embargo, Android utiliza este modo de funcionamiento obligatoriamente, estando fuera del control de las aplicaciones. Esto tiene como consecuencia que el API de MetaWear no consiga realizar una reconexión con esta aplicación porque intenta conectarse a la misma dirección inicial. Esto ocurre, por ejemplo, si se pierde la conexión por alejarse demasiado los dispositivos, lo que obligaría a cancelar la reconexión e iniciar un nuevo escaneo y una nueva conexión. Cabe mencionar que si ambos dispositivos se emparejasen, cabría esperar que pudiese realizarse la reconexión a pesar de la rotación de direcciones privadas, pero se decidió no explorar esta opción por no estar soportada para los dispositivos de MbientLab..

Una vez implementado lo relativo al *advertising*, el objetivo es lograr establecer una conexión con un cliente que utilice el API de MetaWear. En primer lugar, se procedió a replicar los servicios GATT que exponen estos wearables. Para visualizar la jerarquía de servicios y características presentes en el servidor GATT del dispositivo MetaTracker, se volvió a utilizar la aplicación nrf Connect. En la web del SIG de Bluetooth se pueden consultar los UUIDs de servicios y características[33] adoptados por este organismo para multitud de casos de uso comunes. La aplicación nrf Connect reconoce aquellos UUIDs definidos por el SIG y muestra sus nombres automáticamente. Esto no quita que se pueda optar por utilizar UUIDs propios para casos de uso que así lo requieran. Como se puede ver en la figura 5.6b (página 39) estos dispositivos exponen 5 servicios. Dos de ellos son obligatorios según la especificación[14] para cualquier perfil basado en GATT:



(a) *Advertising* (b) Servicios GATT (c) Servicio MetaWear (d) Servicio device information

Figura 5.6: Visualización de *advertising* y servidor GATT de un dispositivo MetaTracker utilizando nrf Connect

- **Generic Access:** contiene información general sobre el dispositivo, como el nombre del dispositivo o parámetros preferidos para establecer la conexión.
- **Generic Attribute:** puede contener la característica opcional Service Changed, que permite notificar al cliente si cambia la estructura del servidor, para que vuelva a realizar el descubrimiento de los servicios.

Cuando utilizamos el API del *Framework* para iniciar un servidor GATT, Android se encarga automáticamente de añadir estos dos servicios. Además, el dispositivo expone otros tres servicios:

- **Battery Service:** expone el estado de la batería del dispositivo. En el dispositivo analizado contiene la única característica que no es opcional para este servicio, la del nivel de batería.
- **Device Information:** expone información del fabricante del dispositivo. En este caso están presentes características para el nombre del fabricante, número de modelo, número de serie, revisión de hardware y revisión de firmware.
- **MetaWear service:** servicio personalizado por el fabricante, único de los aquí descritos que no está adoptado por el SIG de Bluetooth. Contiene dos características, cada una de ellas utilizada para un sentido de la comunicación. El API utiliza la escritura sobre una de ellas para enviar los comandos al dispositivo y activa las notificaciones en la otra para de esta forma recibir los datos producidos por este.

Tal y como se ha descrito en el apartado de diseño, se separó la responsabilidad de definir los servicios específicos que debemos añadir al servidor GATT, así como el comportamiento ante escrituras o lecturas de características, de la clase encargada de gestionar el servidor GATT. En esta iteración se implementó una única clase para el comportamiento de la placa, centrándonos únicamente en imitar el dispositivo MetaTracker.

Una vez replicados estos servicios en nuestro servidor GATT, y respondiendo ante la lectura de las características con los mismos valores observados, todavía no era posible establecer una conexión con el API de MetaWear. Esto ocurre porque al iniciar la conexión, además de leer las características del servicio Device Information, el API envía una secuencia de comandos para descubrir qué módulos soporta la placa y algunos detalles de configuración de éstos. El dispositivo responde a cada comando con la información del módulo correspondiente. Finalmente, el API envía un comando que indica que la conexión se ha establecido, a lo que el dispositivo debe responder adecuadamente. Para implementar las respuestas al descubrimiento de los módulos fue de gran ayuda haber encontrado en el código fuente del API tales conjuntos de respuestas para gran parte de los modelos de placas de MbientLab.

Cuando el servidor GATT recibe una solicitud de escritura de una característica, delegamos en la clase Board. Ésta comprueba si se trata de la característica que utiliza el API para enviar comandos. Si es así, se comprueba si el comando se corresponde con alguno para el descubrimiento de módulos o establecimiento de la conexión y se envía la respuesta correspondiente. Para enviar las respuestas se lanza una notificación sobre la otra característica del servicio MetaWear, que el API configura para activar las notificaciones nada más iniciar la conexión.

Hasta la versión 4.1 de la especificación de Bluetooth, se imponían varias restricciones respecto a la topología de las conexiones. Una de estas limitaciones era que un dispositivo con el rol de periférico puede conectarse únicamente con un dispositivo con el rol central[34]. A pesar de que en la especificación 4.1, con la que es compatible el dispositivo analizado, se eliminaron estas restricciones en cuanto a combinación de roles, se comprobó que el dispositivo MetaTracker deja de realizar *advertising* al establecer la conexión, con lo que permite una única conexión al mismo tiempo. Se replicó este comportamiento deteniendo el *advertising* cuando se realiza la conexión y volviendo a iniciarlo cuando se termina. Hay que tener en cuenta que este comportamiento podría ser susceptible de cambiar en nuevos modelos que utilicen otros SoCs.

Una vez conseguido el establecimiento de la conexión, se continuó con la implementación de la clase ServiceStatus y de la parte de la interfaz en la actividad principal que observará los cambios en ésta y los mostrará en pantalla. En este punto, no se encontró manera de obtener el nombre del dispositivo que se conecte a nuestra aplicación utilizando la conexión existente.

Android dispone de un método para obtener el nombre de un dispositivo remoto pero, según la documentación, el nombre se obtiene durante la realización del escaneo y se guarda en caché. Como en nuestro caso nuestro dispositivo ejerce el rol de periférico, éste no realiza ningún escaneo previo a la conexión, por lo que dicho método no devuelve el nombre. Como solución a esto, se implementó la clase RemoteDeviceGatt para realizar una conexión como cliente y así leer la característica Device Name del servicio Generic Access del dispositivo remoto.

En cuanto a la lectura del RSSI, no se tuvo que añadir nada a la implementación, ya que nuestro dispositivo no interviene en esta lectura. Una vez establecida la conexión, en el dispositivo remoto el API se encarga de obtener las mediciones de este valor.

La lectura del nivel de batería se realiza a través del servicio estándar Battery Service, adoptado por el SIG de Bluetooth. Por lo tanto, de forma análoga a lo implementado para el servicio Device Information, se añadió un caso para las solicitudes de lectura de la característica Battery Level, respondiendo con el nivel de batería que reporte nuestro sistema.

Cabe mencionar que una parte significativa del tiempo de desarrollo de esta iteración se dedicó a averiguar el comportamiento de la comunicación entre el dispositivo MetaTracker y el API. Por un lado, se utilizó la capacidad de los dispositivos Android de capturar el tráfico bluetooth, para luego analizarlo utilizando la herramienta Wireshark. Por otro lado, se estudió el código fuente del API de MetaWear para Android, el cual se encuentra disponible en Github[35]. Una vez conseguido establecer la conexión, también se aprovechó la aplicación resultante del tutorial de MbientLab para ir haciendo pruebas con los distintos comandos del API.

Pruebas

Para la implementación de las pruebas de unidad, se utilizaron varias librerías comúnmente utilizadas para este fin en Android. Se utilizó Junit 4 como marco de trabajo para definir las pruebas y realizar las aserciones. La librería Mockito facilitó la creación de *mocks* y *stubs* para centrarse en probar un único componente cada vez. También se utilizó Roboelectric, que permite ejecutar las pruebas localmente simulando algunas dependencias de la plataforma Android. De no ser así, estas pruebas tendrían que ejecutarse en un dispositivo real o un emulador, lo que es más lento.

Para las pruebas de integración, y al ser la comunicación BLE una parte fundamental de la funcionalidad a probar, se decidió utilizar la capacidad para la realización de pruebas de la aplicación nRF Connect. Esta permite la automatización de pruebas sobre dispositivos BLE, definiendo una suite de pruebas en un archivo *xml*. Para la ejecución de la suite se proporciona un script, que requiere de un teléfono con la aplicación instalada y conectado a través

de usb. Entre las pruebas que permite esta aplicación, están el escaneo filtrando por UUID de servicio, la conexión al dispositivo previamente escaneado, comprobar la presencia de servicios, características y descriptores, comprobación de las propiedades de las características, lectura/escritura de características y recibir notificaciones. Se desarrolló una suite de pruebas que compruebe la correcta implementación del servidor GATT acorde al dispositivo MetaTracker utilizado como referencia y los casos de uso desarrollados en esta iteración. También se fueron realizando pruebas manuales durante todo el proceso de desarrollo, utilizando tanto las aplicaciones oficiales de MbientLab, como la aplicación resultante del tutorial Freefall.

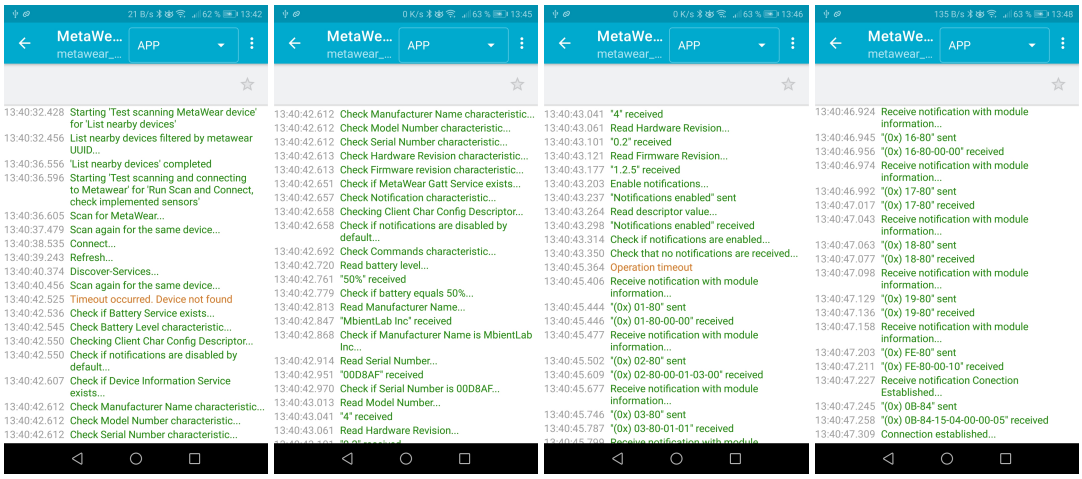


Figura 5.7: Ejecución de tests automatizados a través de nRF Connect

Un problema encontrado durante esta iteración, fue que sólo se establecía la conexión tras varios reinicios de la aplicación. El problema era que no se añadían correctamente todos los servicios de manera repetible, y cuando no están presentes algunas características del servicio Device Information o del servicio MetaWear, el API no puede establecer la conexión. En un primer momento, se pensó que podría ser algún defecto del dispositivo concreto que estábamos utilizando o que quizás el stack de bluetooth en la versión utilizada no era lo suficientemente estable/maduro. También se probó a eliminar el servicio de la batería, prescindible para el establecimiento de la conexión, para comprobar si se apreciaba alguna mejoría. Esto resultó en una pequeña mejoría, requiriéndose menos intentos para lograr la conexión pero sin ninguna fiabilidad. No fue hasta transcurrido parte del desarrollo de la siguiente iteración que se encontró una solución a este problema, cuando se probó a introducir varios tiempos de espera entre las llamadas proporcionadas por el sistema Android para añadir los servicios GATT. La documentación de estas llamadas es mejorable, pues no se encontró ninguna indicación de que éstas sean asíncronas, ni recomendación alguna al respecto. Con esto, se logró un funcionamiento mucho más estable, estableciéndose la conexión todas las veces probadas.

Una vez finalizada la iteración y probado el resultado de esta, se obtuvo un prototipo parcialmente funcional que se utilizó en una reunión con el cliente. De esta manera, se pudieron tener en cuenta sus valoraciones de cara a la siguiente iteración.

5.3.2 Iteración 2

Análisis

Requisitos funcionales:

- Capacidad de realizar *streaming* tanto de datos de aceleración del dispositivo sin procesar como de cambios de orientación, y configuración del módulo del acelerómetro.
- Utilizar el acelerómetro del dispositivo que ejecuta la app para producir los datos que se envían.
- Mostrar en pantalla el estado del módulo del acelerómetro.
- Capacidad de devolver un valor de temperatura controlado por el usuario.
- Capacidad de devolver un valor de frecuencia cardíaca controlado por el usuario.
- Permitir seleccionar entre varios modelos de placa.
- Permitir que el usuario controle también el nivel de batería leído.

Requisitos no funcionales:

- Favorecer la extensibilidad de las funcionalidades en el diseño; para añadir en el futuro más sensores y modelos de placas.

Los casos de uso detectados para la segunda iteración pueden verse en la figura 5.8 (página 44). El usuario inicia la aplicación y escoge un modelo de placa, lo que inicia el servicio en segundo plano y abre la actividad principal. En esta actividad el usuario podrá modificar los valores de temperatura, frecuencia cardíaca y nivel de batería a través de controles en la interfaz. La acción de modificar los valores no implica que se envíe nada, ya que éstos se envían en respuesta a una solicitud de lectura realizada por parte del API de MetaWear. De forma similar, el encendido/apagado del acelerómetro, inicio/detención del *streaming* de datos del acelerómetro, inicio/detención de la notificación de cambios de orientación y la configuración de rango y frecuencia de sampleo se realizarán de forma automática en respuesta

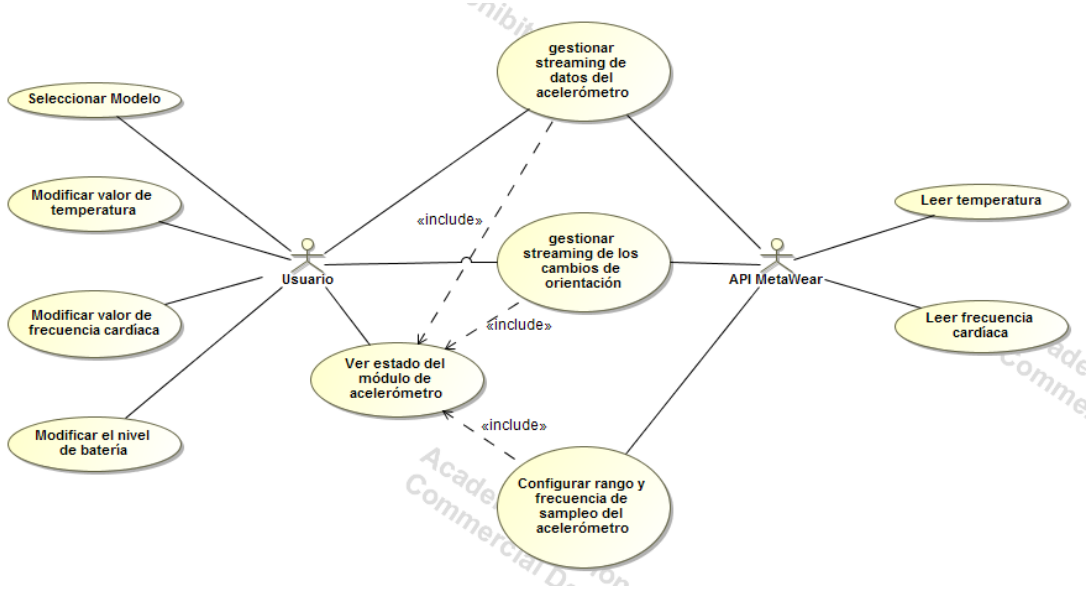


Figura 5.8: Casos de uso de la segunda iteración

a los comandos correspondientes recibidos del API. El usuario podrá visualizar en la misma actividad el estado de estos componentes. Cuando se haya iniciado el *streaming* de datos del acelerómetro, el usuario podrá mover el dispositivo para producir los datos de aceleración deseados. Análogamente, mientras esté activado el envío de la orientación, el usuario podrá girar el dispositivo para producir una notificación cuando ésta cambie.

Diseño

El interfaz `Board` define los métodos que utiliza el servidor GATT para delegar cuando se produce una solicitud de lectura o escritura de una característica. También se definen en este interfaz métodos para obtener los servicios y características que se deben exponer en el servidor y el UUID de servicio que se incluirá en el *advertising*.

La clase abstracta `MetaWearBoard` implementa el interfaz `Board` y encapsula las peculiaridades propias de los dispositivos MetaWear que son comunes a todos los modelos. Se definen también una serie de métodos abstractos que deberán ser implementados por las clases que hereden de ésta, y que representarán a modelos concretos de estos dispositivos. Se definen en esta clase todos los servicios y características que se añadirán al servidor GATT, y se devuelve el UUID del servicio propio de MetaWear para que sea incluido en el *advertising*.En cuanto a los comandos recibidos por el API, los relativos al proceso de conexión se procesarán en esta clase, mientras que el resto lo harán las clases concretas de cada modelo utilizando el patrón método plantilla. También se responderá a las lecturas de características en esta clase,

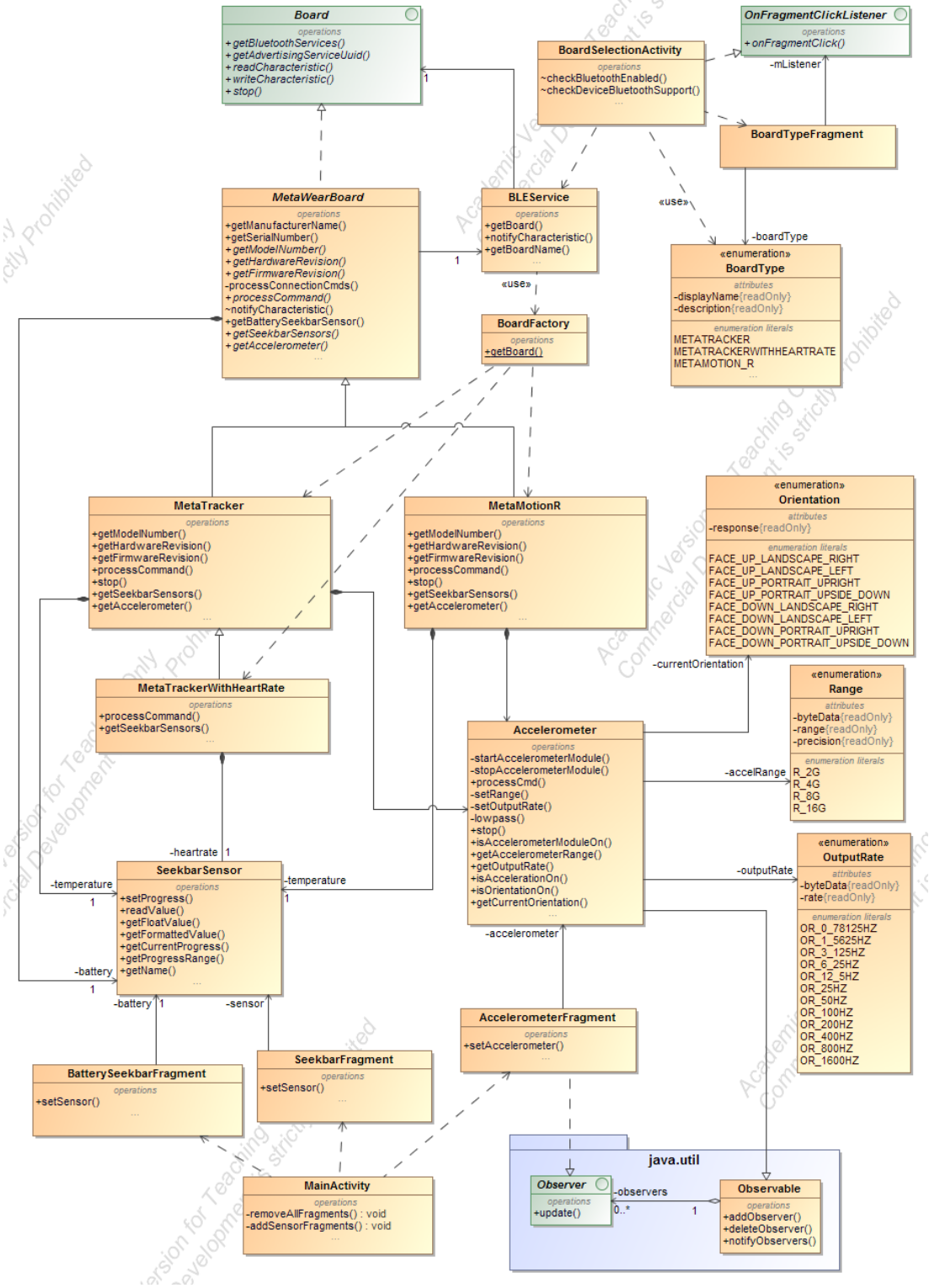


Figura 5.9: Diagrama de clases que muestra la jerarquía de placas y sensores

utilizando el mismo patrón para que cada subclase pueda devolver aquellas que son específicas para cada modelo: número de modelo, revisión de hardware y revisión de firmware.

Se ha utilizado el patrón método factoría parametrizada usando genericidad para la creación de la clase correspondiente a la selección que realice el usuario. De esta manera, la clase `BLEService` depende únicamente del interfaz `Board`, y su funcionamiento es independiente del tipo de placa escogida. Añadir nuevos modelos sería sencillo, teniendo que crear una subclase de `MetaWearBoard`, similar a las ya implementadas `MetaTracker` y `MetaMotionR`. También se podrían añadir clases que hereden de los modelos ya implementados, lo que permitiría modificar la configuración de sensores o el comportamiento definido para estos modelos. Esto es lo que se ha decidido hacer con la clase que llamamos `MetaTrackerWithHeartRate`, que es una subclase de `MetaTracker` y únicamente añade un sensor para la frecuencia cardíaca. Esta configuración se ha creado con el objetivo de desarrollar la aplicación ControlFlu, que requería la monitorización de este parámetro, además de los ya implementados en la clase `Metatracker` para temperatura y acelerómetro. Finalmente, tendríamos que definir un nuevo valor del enumerado `BoardType` para que se muestre una nueva entrada en la pantalla de selección.

Al separar las particularidades de los dispositivos MetaWear en la clase `MetaWearBoard`, sería relativamente sencillo extender la funcionalidad de la aplicación para emular otro tipo de dispositivos que hagan uso del rol de servidor GATT. Este no es un objetivo de este proyecto, pero se considera apropiado facilitar la extensibilidad futura al no suponer un aumento considerable de la complejidad. Tanto `BLEService` como `GattServer` delegan en el interfaz `Board` el comportamiento ante las operaciones GATT, por lo que es responsabilidad de las clases implementadoras toda la lógica en respuesta a estas operaciones. Para definir otro tipo de dispositivos habría que crear una nueva clase que implemente el interfaz `Board`, definiendo los servicios y características que quiere exponer en el servidor GATT, así como el UUID de servicio si lo desease, y el procesamiento de solicitudes de lectura y escritura de características.

La actividad `BoardSelectionActivity` muestra un fragment por cada valor de `BoardType`. Para la comunicación entre fragments y la actividad, tal y como recomienda la documentación de Android, `BoardSelectionActivity` implementa el interfaz `OnFragmentClickListener` definido en `BoardTypeFragment`.Cuando el usuario pulse alguno de los fragments, éste notificará a la actividad pasando el valor que corresponda del enumerado `BoardType`. Finalmente, `BoardSelectionActivity` inicia el servicio pasando el valor seleccionado y seguidamente provoca que naveguemos a nuestra actividad principal.

La actividad principal se enlaza con el servicio y obtiene una referencia a `Board`. A con-

tinuación, obtiene los sensores presentes en la placa y en función de estos crea los fragments que se encargarán de la interfaz relativa a cada uno de ellos. Igual que se hizo en la primera iteración para mostrar el estado del servicio BLE, hemos utilizado el patrón observador para que `AccelerometerFragment` muestre el estado del módulo del acelerómetro.

Implementación

Después de haber conseguido el establecimiento de la conexión en la iteración anterior, se pudieron ir registrando todas las peticiones de escritura a la característica que utiliza el API para enviar los comandos a nuestro servidor GATT. De esta forma, se fueron probando todas las llamadas al API relacionadas con los casos de uso de la aplicación y registrando los valores de los comandos producidos. El API organiza las funcionalidades en distintos módulos, y se pudo comprobar que el primer byte de cada comando siempre identifica a qué módulo pertenece.

El caso de uso que se implementó en primer lugar, fue el del *streaming* de los datos del acelerómetro, por su mayor riesgo, complejidad e importancia en la aplicación. Todos los dispositivos de MbientLab actualmente disponibles utilizan el IMU BMI160 de Bosch, que combina acelerómetro y giroscopio. Por este motivo los comandos que se implementaron son los correspondientes al módulo del acelerómetro específico para este modelo de IMU. Este módulo del API puede producir diferentes tipos de datos, siendo de interés en este caso dos de ellos: los datos crudos de aceleración en los tres ejes y los cambios de orientación del dispositivo. Otros datos derivados que puede producir serían el número de pasos, la detección de movimiento significativo o si el dispositivo se encuentra sobre una superficie horinzontal como una mesa, entre otros.

Toda la funcionalidad relativa al módulo del acelerómetro se implementó en la clase `Accelerometer`, de modo que las clases de los modelos de placa puedan incluirla como uno de los sensores disponibles. En este caso, esas clases deberán pasar a esta los comandos relativos al acelerómetro, comprobando su primer byte. Teniendo en cuenta que la mayoría de teléfonos móviles hoy en día disponen de un acelerómetro, se decidió utilizar el sensor del teléfono para producir los datos que se envíen al cliente. En Android se dispone de la clase `SensorManger` para acceder a los sensores presentes en el dispositivo y registrar callbacks para recibir los datos. Una vez obtenidos los datos de aceleración en un callback, solo resta enviarlos a través de una notificación en el formato adecuado. Para averiguar cómo se componen los mensajes que contienen los datos crudos de aceleración, se analizó en Wireshark tráfico previamente capturado con la placa MetaTracker (ver figura5.10, página 48). Comprobamos que los valores de todas las notificaciones comienzan con los mismos 2 bytes. El primero, al

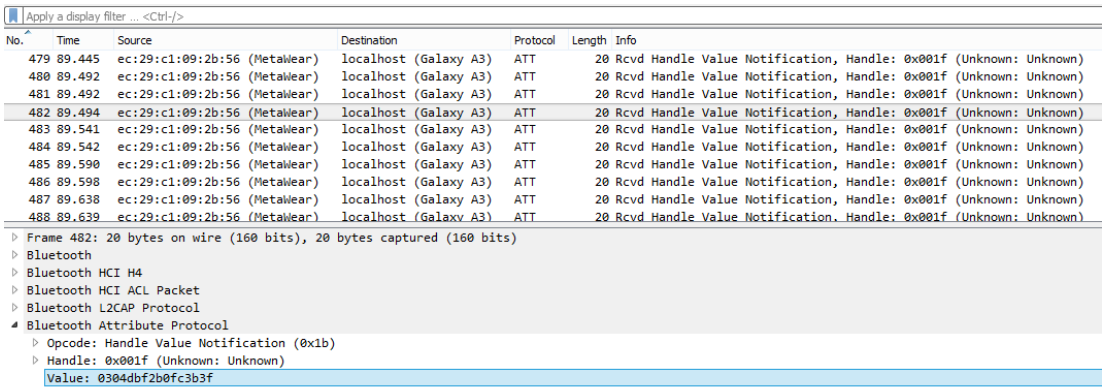


Figura 5.10: Notificaciones con los datos de aceleración visualizados con Wireshark

igual que se había visto con los comandos, hace referencia al módulo; en este caso 0x03 identifica al módulo del acelerómetro. El segundo byte en este caso indica el tipo de dato que se está enviando, 0x04 para los datos de aceleración; y los seis bytes siguientes se corresponden con los datos, dos bytes por la aceleración en cada eje. Hay que mencionar que al ser la longitud de los datos la misma independientemente del rango de aceleración configurado, esto significa que a mayor rango la precisión de los datos será menor.

Este módulo también permite la configuración tanto del rango de aceleración como de la frecuencia de sampleo. La estructura del comando que escribe la configuración en la placa sigue la estructura mostrada en el ejemplo de la figura 5.11 (página 49). Ambos se configuran con un único comando y utilizan un byte para cada uno de ellos conforme a una serie de valores predefinidos. El IMU BMI160 permite elegir entre 12 frecuencias de sampleo, siendo la menor 0.78125Hz y la mayor 1600Hz; y entre 4 rangos, entre 2g y 16g. Para implementarlo, se optó por definir un enumerado para cada uno de los parámetros, con su correspondiente código para cada uno de los valores. En el enumerado del rango, además, se incluyó un campo para la precisión utilizada en los datos de aceleración, ya que ésta siempre debe ser consistente con el rango configurado. Por otra parte, las frecuencias de sampleo disponibles dependerán del acelerómetro que posea el dispositivo que ejecute la aplicación. Al registrar los callbacks de los sensores, se solicita el valor deseado de frecuencia de sampleo y Android notifica los datos a la frecuencia válida más cercana.

BLE transmite los paquetes en pequeñas ráfagas con una frecuencia definida por el *Connection Interval*, que según el estándar puede ser desde 7.5 ms hasta 4 s. Entre estas ráfagas, se apaga la radio para ahorrar batería. Para conseguir una transmisión de datos del acelerómetro más fluida, se utilizó la conexión como cliente GATT que se implementó anteriormente en la clase `RemoteDeviceGatt` para solicitar el valor mínimo posible de *Connection Interval*.

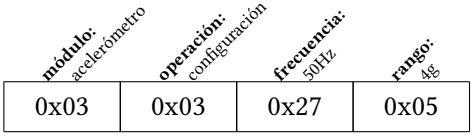


Figura 5.11: Ejemplo de comando de configuración de frecuencia y rango del acelerómetro

La tasa de transferencia que ofrece BLE no permite el *streaming* en tiempo real de frecuencias de sampleo muy elevadas, tal y como indica MbientLab en el apartado de sensores de su FAQ[36]. La configuración de frecuencias de sampleo elevadas en el módulo del acelerómetro de las placas reales, solo tiene sentido combinada con la capacidad de *log* en su memoria interna, para obtener los datos posteriormente. El mecanismo de *log* no forma parte del alcance de este proyecto, por lo que el uso de frecuencias de sampleo elevadas no resulta práctico con la aplicación que se ha desarrollado por la congestión que producen sobre la comunicación. Con los teléfonos de los que se dispuso para el desarrollo, la fluidez de la comunicación empeoraba con frecuencias de sampleo superiores a 50Hz para los datos de aceleración.

Otro tipo de datos que puede producir el IMU BMI160 son los cambios de orientación. Define ocho orientaciones posibles en función de la inclinación del dispositivo y solo lanza notificaciones en el momento en el que un cambio de esta significan pasar de una posición a otra. Para obtener en Android los ángulos de inclinación de nuestro dispositivo, es necesario que este disponga, además de acelerómetro, de un magnetómetro. Una vez obtenidos los ángulos de inclinación se puede comprobar en qué posición se encuentra en cada momento, y mandar una notificación cuando se produzca un cambio. De forma similar a lo descrito para los valores de configuración, también se definió un enumerado con las 8 posiciones posibles y el código que identifica a cada uno en la notificación. Un problema encontrado en este momento fue que se lanzaban muchos cambios de orientación no deseados, incluso en ocasiones de forma errática ante ciertos movimientos bruscos sin cambio de orientación. Para mitigar este efecto, se implementó un filtro paso bajo para obtener una versión suavizada de los datos de aceleración, que utilizamos en la función del sistema que calcula los ángulos de inclinación. Experimentando con este filtro, se obtuvo un resultado satisfactorio.

Además de los comandos para iniciar y detener el envío de cada tipo de datos que soporta el módulo, el API dispone de comandos para iniciar y detener el módulo de forma global. El inicio del módulo debe ser lo último que se hace tras la configuración e inicio de los tipos de datos deseados, tal como indica la documentación de MbientLab[37]. En la implementación de nuestra aplicación, se registran los callbacks de los sensores al recibir el comando de inicio global, y se eliminan los callbacks con el comando de parada global, con el fin de reducir el consumo elevado de batería que acarrean los sensores siempre que no se haga uso de ellos.

A diferencia de lo que ocurría con el acelerómetro, en el caso de la temperatura solo se

producen datos en respuesta a solicitudes de lectura. Por lo tanto, si el cliente desea obtener lecturas periódicamente, deberá programar dichas solicitudes. El API también dispone de un módulo `Timer` que permite la ejecución de comandos periódicamente en la placa, que también serviría para este propósito. El comando que se produce al solicitar una lectura de temperatura sigue el mismo patrón observado anteriormente. El primer byte indica el módulo de temperatura y el segundo el tipo de operación, una lectura. El tercer byte se utiliza para identificar cual de los sensores de temperatura disponibles se quiere leer.

Todos los modelos de MbientLab incluyen al menos dos sensores para la temperatura: uno que llaman termistor predeterminado y otro incluido en el SoC que desaconsejan utilizar[38]. Además, los modelos que estén equipados con un barómetro de Bosch, también podrán utilizarlo para medir temperatura. Finalmente, también ofrecen acceso a un sensor de temperatura externo, que tendría que conectarse a los pines GPIO. Para este caso de uso, se tomó la decisión de implementar un único sensor de temperatura y responder a las lecturas de los diferentes sensores de temperatura considerados en el API con el mismo valor. Por el diseño de la aplicación sería sencillo cambiar este aspecto, y disponer de múltiples sensores de temperatura que pudiesen manipularse independientemente. De forma similar a lo descrito anteriormente para el acelerómetro, se revisaron las capturas de tráfico para observar el formato de las respuestas. Los tres primeros bytes son iguales a los recibidos en la solicitud, y se utilizan dos bytes adicionales para enviar el valor leído. Se pudo comprobar que la precisión de los datos que se pueden enviar para este sensor es de 0.125ºC.

A continuación se creó la clase `SeekbarSensor`, para representar aquellos sensores que podremos manipular en la interfaz a través de un *slider*. Se encarga de almacenar el valor actual del sensor, así como su nombre, precisión, su rango aceptable de valores y el formato con el que queremos que se impriman sus valores en pantalla. De esta forma podemos crear un componente de la interfaz genérico, `SeekbarFragment`, que únicamente se encarga de aquello relativo a la interfaz acorde a la especificación del `SeekbarSensor` asociado.

Una vez implementado el fragment `SeekbarFragment`, fue sencillo implementar otro fragment muy similar para permitir el control del valor de batería, BatterySeekbarFragment. El único motivo para crear otro fragment para esta tarea fue que se deseaba poder elegir qué dato enviar ante las lecturas de batería: el seleccionado manualmente o el nivel actual de batería del sistema.

También se implementó el fragment `AccelerometerFragment`, que se encarga únicamente de mostrar el estado del módulo del acelerómetro, y la lógica en `MainActivity` necesaria para crear dinámicamente los fragments adecuados en función de los sensores de la placa escogida por el usuario.

Se creó la clase `MetaTrackerWithHeartRate`, heredando de `MetaTracker`. Esta clase mantiene una referencia a otro `SeekbarSensor` para la frecuencia cardíaca, y sobreescribe el método `processCommand()` para responder a la lectura de este sensor. Como ninguno de los modelos de placa disponibles poseen sensor de frecuencia cardíaca, el API de MetaWear no dispone de un módulo específico para ello. Sin embargo, sí dispone de un módulo para leer las entradas GPIO de las que están dotadas estas placas, y se presupone la viabilidad de utilizar alguno de los sensores de frecuencia cardíaca que se encuentran en el mercado a través de estas entradas (esta posibilidad se menciona en la FAQ del fabricante[36]). Por lo tanto, se decidió responder a la lectura de uno de los pines GPIO con el valor de frecuencia cardíaca que ajuste el usuario. Hay que tener en cuenta que esto es una simplificación, y que emular de manera más realista la señal producida por un sensor de este tipo se consideró fuera del alcance de este proyecto.

Finalmente, se pulieron algunos aspectos gráficos y se añadió una animación que indica que el dispositivo está realizando *advertising*. También se añadieron botones para incrementar y decrementar de forma precisa los valores en `SeekbarSensor`, ya que el *slider* resultaba incómodo para incrementos pequeños.

Pruebas

Al igual que en la primera iteración, se utilizaron las bibliotecas JUnit 4, Mockito y Roboelectric para la implementación de las pruebas de unidad.

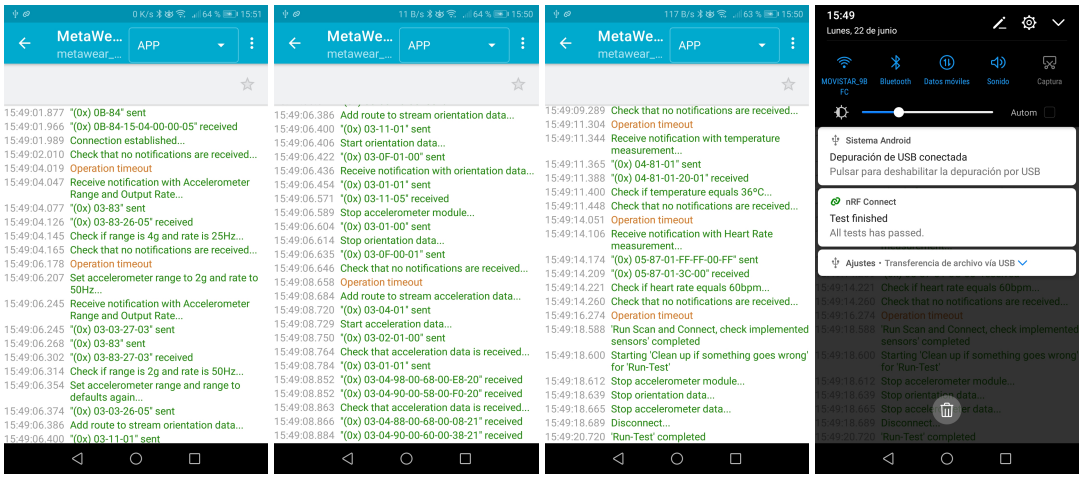


Figura 5.12: Casos de prueba añadidos en la segunda iteración, ejecutados con nRF Connect

En cuanto a las pruebas de integración, durante esta iteración se amplió la suite de pruebas realizada en la primera iteración con la herramienta nRF Connect. Con ello, se pudieron

automatizar pruebas que comprueben que los sensores envían datos, así como cambiar la configuración del módulo del acelerómetro. Sin embargo, con esta herramienta no se pudieron desarrollar casos de prueba que comprueben que los datos producidos sean los que el usuario desea. Para validar esto y el correcto funcionamiento general de la aplicación, se realizaron pruebas manuales de todos los casos de uso desarrollados, utilizando la aplicación de ejemplo de MbientLab y el resultado del tutorial que se ha modificado para facilitar este fin.

5.4 ControlFlu

5.4.1 Iteración 1

En esta iteración se aborda lo relativo a la conectividad con el dispositivo wearable y la obtención de los datos pertinentes a través de un servicio en segundo plano.

Análisis

Requisitos funcionales:

- Comprobaciones relativas a la utilización de Bluetooth Low Energy.
- Escanear dispositivos MetaWear.
- Conexión a un dispositivo MetaWear previamente escaneado.
- Recibir datos de aceleración en los tres ejes.
- Recibir datos de orientación.
- Leer datos de temperatura.
- Leer datos de frecuencia cardíaca.
- Leer nivel de batería.
- Leer RSSI.
- Mostrar en una barra inferior los valores actualizados de los datos leídos.
- Reconexión automática cuando se pierda la conexión, mostrando un diálogo al usuario.

Requisitos no funcionales:

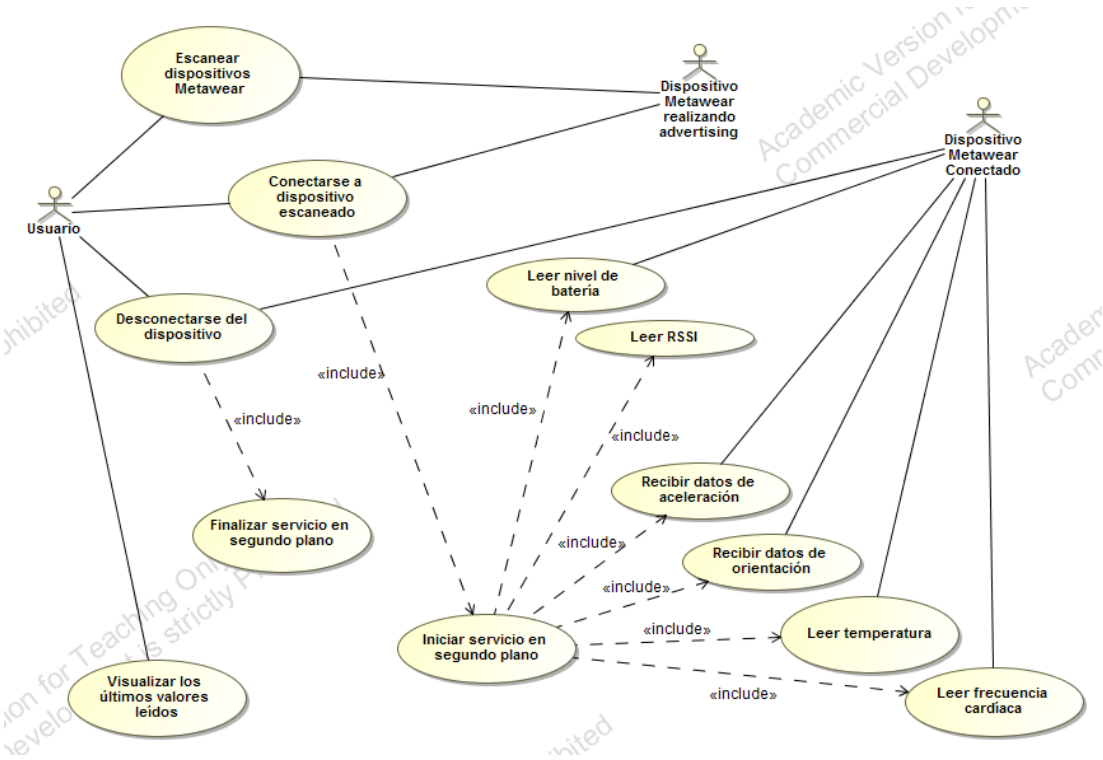


Figura 5.13: Casos de uso de la primera iteración

- Comunicarse con el wearable en un servicio en segundo plano.
- Información actualizada en tiempo real, el usuario debe percibir una latencia baja.
- Diseño que facilite añadir nuevas alertas, lectura de más sensores, etc.

En la figura 5.13 podemos ver los casos de uso correspondientes a la primera iteración de esta aplicación. Cuando el usuario inicie la aplicación, verá una pantalla donde se muestra la lista de dispositivos MetaWear encontrados en el escaneo. El escaneo se iniciará automáticamente, pero también puede ser iniciado por el usuario o detenido antes de que termine. A continuación, el usuario podrá pulsar alguno de los dispositivos de la lista para iniciar una conexión. Si se establece la conexión correctamente, se abrirá la pantalla principal de la aplicación, donde se mostrarán los datos de los sensores en tiempo real. Si el usuario solicita explícitamente retroceder en el flujo de navegación, se procederá a la desconexión con el dispositivo para volver a la pantalla de escaneo. Aparecerá un diálogo de confirmación para evitar desconexiones accidentales que empeorarían la experiencia del usuario.

Al establecerse la conexión también se iniciará el servicio en segundo plano que se encargará de obtener los datos utilizando el API de MetaWear. De esta manera, aunque el usuario bloquee su teléfono o abra otras aplicaciones el servicio continuará recibiendo los datos.

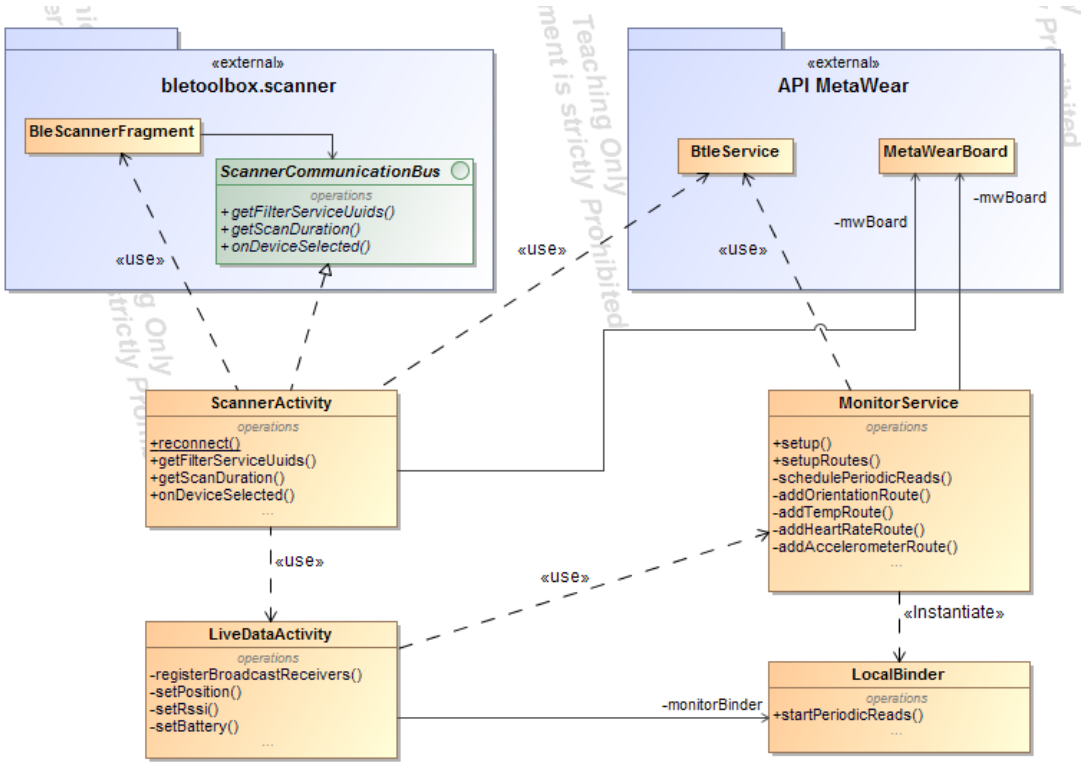


Figura 5.14: Diagrama de clases de la primera iteración

La obtención de los datos se iniciará automáticamente sin intervención del usuario, siempre que mantengamos una conexión con el dispositivo wearable. Para los datos de aceleración y orientación se configurará el *streaming* para que la placa envíe los datos según se vayan produciendo. Para el resto de datos, el servicio realizará las lecturas de forma periódica. Al volver a la pantalla de escaneo, también se procederá a detener el *streaming* de los datos y a terminar la ejecución del servicio en segundo plano.

En la parte inferior de la pantalla, el usuario podrá visualizar las últimas lecturas de temperatura y frecuencia cardíaca, así como un icono que indicará la posición del paciente. También se mostrará la intensidad de la señal recibida y el nivel de batería del dispositivo wearable.

En caso de que se pierda la conexión, la aplicación tratará de reconectarse automáticamente y se mostrará un diálogo mientras no se recupere la conexión. El diálogo ofrecerá la opción de cancelar la reconexión.

Diseño

Para realizar el escaneo de los dispositivos MetaWear cercanos, se utilizó una biblioteca de MbientLab, la misma que se utiliza en la aplicación de ejemplo que proporcionan en su web. Se añadió el fragment `BleScannerFragment` a la actividad, y se hizo que esta implemente el interfaz `ScannerCommunicationBus`. Este interfaz permite pasarle al fragment la duración del escaneo y los UUIDs de servicio para realizar un filtrado, y así no mostrar al usuario otro tipo de dispositivos BLE cercanos. Cuando el usuario seleccione un dispositivo, el fragment devolverá el dispositivo a través de otro método de este interfaz.

Para establecer la conexión y llevar a cabo la comunicación con el dispositivo wearable, se utilizó el API que MbientLab pone a disposición de los desarrolladores[35]. Para acceder a este API, se debe enlazar algún componente de la aplicación con el servicio `BtleService`, para así poder obtener una referencia a `MetaWearBoard`, clase central del API. Esta representa al dispositivo wearable, y expone los módulos disponibles y las operaciones que se necesitan para la comunicación con él. En esta aplicación se enlazan con el servicio del API dos componentes: `ScannerActivity` y `MonitorService`. La conexión se realiza en `ScannerActivity`, cuando el fragment notifica el dispositivo que pulsó el usuario. De esta forma, sólo en caso de que se establezca correctamente la conexión se inicia el servicio en segundo plano y se produce la navegación hacia la pantalla `LiveDataActivity`. Además de encargarse de iniciarlo, `ScannerActivity` también detiene el servicio cuando el usuario retroceda a esta pantalla, volviendo a iniciar el proceso de escaneo.

El servicio en segundo plano se encarga de utilizar el API de MetaWear para configurar los sensores cuando sea necesario y realizar las lecturas. También se encarga de enviar los datos recibidos al resto de componentes de la aplicación. Para la comunicación entre `LiveDataActivity` y `MonitorService` se utilizaron dos mecanismos distintos. Por un lado, Android proporciona el mecanismo de *broadcast* que permite el envío de intents entre componentes de la misma aplicación o incluso entre distintas aplicaciones. Estos intents sirven para notificar algún tipo de evento, y también permiten incluir objetos serializables como extras. Este mecanismo es conveniente y proporciona algunas ventajas, como evitar un alto acoplamiento entre los componentes que se envían mensajes. Se utilizaron estos broadcasts para enviar desde el servicio los datos según se vayan recibiendo de la placa. También para notificar a la actividad cuando se produzca una desconexión inesperada y la consiguiente reconexión. En sentido contrario, la actividad notifica al servicio si el usuario decide provocar la desconexión.

En un primer momento, se decidió utilizar los broadcast para toda la comunicación e interacción entre el servicio y el resto de componentes de la aplicación, utilizando un servi-

cio iniciado sin enlazarse a otros componentes. Esto ofrecería un menor acoplamiento entre estos componentes. Sin embargo, en la siguiente iteración, y una vez ya implementado con broadcasts lo descrito anteriormente, se decidió implementar el enlazado del servicio para una interacción más directa y poder utilizar el patrón observador para la notificación de alertas. También se supone que esta comunicación es más eficiente que el uso de broadcasts, por lo que se pensó en refactorizar lo implementado, considerando especialmente el caso del envío de los datos de aceleración. En este caso, al no utilizar broadcasts para ninguna comunicación fuera de la aplicación, se pudo utilizar una implementación más eficiente que proporciona Android para broadcasts locales. Todas las pruebas que se realizaron indicaron que el uso de broadcasts no suponía un cuello de botella para esta aplicación, al menos con las limitaciones actuales en la tasa de transferencia de BLE. Por este motivo, se consideró que no estaba justificada la refactorización, aunque podría mejorar la coherencia del código.

Para que otros componentes puedan enlazarse al servicio, se necesitó definir una clase que herede de la clase `Binder`, en este caso `LocalBinder`, y que será la que el sistema devuelva a los componentes que quieran enlazarse. En esta clase será donde se definan las operaciones que sirven para interactuar con el servicio.

Implementación

En primer lugar, se incluyeron las dependencias de compilación en el proyecto de Android para las bibliotecas de MetaWear y de la herramienta para el escaneo. También se declaró el servicio `BtleService` de MetaWear en el archivo AndroidManifest.xml del proyecto para que los componentes de la aplicación puedan utilizarlo.

Para poder conectarse posteriormente al dispositivo, se realiza la llamada para enlazarse a `BtleService` al inicio del ciclo de vida de la actividad de escaneo, en su método `onCreate()`. A través de un callback, Android notifica que se ha realizado el enlace devolviendo un objeto `Binder`.

Para utilizar la biblioteca que realiza el escaneo, se incluyó el fragment en el archivo de layout para la actividad `ScannerActivity` y se implementó el interfaz `ScannerCommunicationBus` en esta. El método `onDeviceSelected()` de este interfaz es el que utiliza el fragment para devolver el dispositivo en el que pulse el usuario. En el cuerpo de dicho método es donde se implementó todo lo que se debe ejecutar en respuesta a dicha acción.

Utilizando el `Binder` que se guardó previamente, se obtiene un objeto `MetaWearBoard` que se utiliza para establecer la conexión. Justo antes de iniciarla, se muestra una ventana de diálogo para informar al usuario de que la conexión está en curso, con un único

botón que le permita cancelarla. El usuario no tiene otra manera de cerrar esta ventana de diálogo. Convenientemente, el fragment de la biblioteca comprueba si el bluetooth está apagado en el sistema y lanza el intent apropiado para solicitar al usuario que lo active.

El API de MetaWear utiliza el framework *Bolts*[39] para manejar las tareas asíncronas, devolviendo objetos de la clase `Task` en todas las operaciones de este tipo. Este es el caso de la conexión, por lo que debemos utilizar los métodos que proporciona *Bolts* para esperar a que termine la tarea. En caso de que la operación tenga éxito, se cierra el diálogo para seguidamente iniciar el servicio `MonitorService` y navegar hacia la actividad principal, `LiveDataActivity`. En caso de que falle el establecimiento de la conexión, y mientras el usuario no cancele la operación, se repite indefinidamente el intento de conexión. Además, en el intent que se utiliza para iniciar el servicio, se pasa como extra el dispositivo que seleccionó el usuario para que, del mismo modo que hizo esta actividad, el servicio pueda obtener el objeto `MetaWearBoard` que lo representa.

Se debe mencionar que buena parte del código de la actividad `ScannerActivity` pertenece a la aplicación de ejemplo para Android de MbientLab[13], que se adaptó ligeramente.

Al inicio del ciclo de vida del servicio `MonitorService`, hacemos que éste se ejecute en primer plano, proporcionando al sistema la notificación que debe mostrarse mientras se ejecute de este modo. Esto es necesario para indicar al sistema que una interrupción del servicio sería perceptible para el usuario, por lo que le otorgará una prioridad alta ante cierres inesperados por falta de recursos.

En `MonitorService`, el proceso de enlazado al servicio de MetaWear y la obtención del objeto `MetaWearBoard` es el mismo que en `ScannerActivity`. Para comenzar a recibir datos de los sensores, primero se obtienen los módulos para los sensores de temperatura, GPIO y acelerómetro. Como ninguno de los modelos disponibles en este momento están equipados con sensor de frecuencia cardíaca, el API tampoco dispone de un módulo para ello. Se utilizó el módulo GPIO para simular la obtención de estos datos en conjunción con la aplicación MetaWear Emulator que se utilizó para el desarrollo. Hay que tener en cuenta que en caso de utilizar algún sensor real conectado a los pines GPIO de alguna de estas placas, posiblemente habría que realizar algún tipo de procesamiento sobre la señal que produzcan estos sensores. También es posible que con nuevos modelos de estos wearables, se añada algún módulo específico para la frecuencia cardíaca.

A partir de cada módulo se obtiene el objeto `DataProducer` adecuado para el tipo de datos que deseamos obtener. A continuación, se le añade a este lo que MbientLab denomina como rutas. Con un builder, podemos ir encadenando varias operaciones que ofrecen cierto procesamiento en la propia placa wearable, para finalmente establecer que se guarden los

datos resultantes en la memoria de la placa o que se envíen a la aplicación en tiempo real. En este caso, no se desea ningún procesamiento en la placa, por lo que se configuraron las rutas de forma que se envíen en tiempo real todos los datos. La operación de la ruta que configura el *streaming* de los datos recibe un `Subscriber`, que se implementó en forma de clases anónimas para definir lo que haremos para cada dato al recibirlo en el servicio. En esta iteración, la implementación se limitó a transmitir los datos a través de broadcasts para que la actividad principal pueda recibirlos y mostrarlos.

En cuanto a la forma en la que se producen los datos, pueden clasificarse en dos grupos. Para los datos de aceleración y orientación, ambos procedentes del módulo del acelerómetro, se llama a los métodos del API que inician la producción de los datos y al método que enciende el acelerómetro. Mientras no se apague, los datos se irán enviando según la frecuencia de sampleo configurada o cuando la orientación cambie. Antes de iniciarlo, también se utiliza el API para configurar la frecuencia de sampleo y el rango del acelerómetro. Con las pruebas realizadas, se consideró suficiente para la detección de convulsiones una frecuencia de sampleo de 25Hz y un rango de 4g. Antes de realizar una desconexión, se apaga el acelerómetro para que el wearable no consuma más batería de la necesaria mientras la aplicación no se comunique con él.

Por otro lado, para la temperatura y GPIO, cuando se llama al método de lectura del API se devuelve un único dato en la ruta previamente configurada. Para conseguir lecturas periódicas para estos datos, se utilizó la clase `Timer` de java.util para programar las lecturas con la frecuencia deseada. También se programaron de la misma manera las lecturas de RSSI y del nivel de batería. Inicialmente, estas lecturas se iniciaban lo antes posible, después de configurar las rutas, pero se pudo observar que en ocasiones no se mostraba la primera lectura en pantalla. Esto se debía a que la actividad principal puede tardar más en iniciarse y registrar su `BroadcastReceiver` de lo que tardaba el servicio en enviar los primeros datos, por lo que podían perderse los primeros valores leídos. Para las lecturas con una frecuencia elevada, la experiencia del usuario no se vería muy afectada; pero para algún sensor con una frecuencia menor podría significar no mostrar ningún valor hasta pasado un tiempo considerable. Para solucionar esto, se añadió un método a `LocalBinder` que permite a la actividad iniciar las lecturas en el servicio una vez está lista para recibir los datos.

El API también permite definir un handler para el evento de una desconexión inesperada. En tal caso, se envía un broadcast a la actividad para que muestre un diálogo como el de la conexión, que le indique al usuario que la aplicación está intentando reconectarse. Igual que para la conexión en `ScannerActivity`, se intenta establecer la conexión repetidamente hasta que tenga éxito o el usuario lo cancele. En caso de que el servicio consiga establecer la conexión, este envía otro broadcast para que la actividad cierre el diálogo. En caso de que el

usuario cancele la reconexión a través del botón del diálogo, es la actividad la que envía un broadcast al servicio para que cancele la operación. A continuación, la actividad se cierra y la aplicación vuelve a la pantalla de escaneo, deteniéndose el servicio.

Para terminar la iteración, se implementó la representación visual de los datos recibidos en una barra inferior de la pantalla principal. Para la posición del paciente, un icono con una flecha muestra en todo momento si está boca arriba o boca abajo. Para la temperatura y la frecuencia cardíaca se muestra el último valor recibido al lado de un icono representativo. También se muestra en el lado derecho de la barra dos iconos que indican el nivel de batería y la intensidad de la señal con el wearable. En caso de que la aplicación se muestre en formato apaisado, se muestran los valores de batería y RSSI al lado de su representación visual.

Pruebas

La implementación de las pruebas de unidad se realizó con la ayuda de las bibliotecas Junit4, Roboelectric y Mockito, como se había hecho para la aplicación MetaWear Emulator.

Como la aplicación depende de la biblioteca de MbientLab para su funcionamiento, para llevar a cabo las pruebas de integración de forma automatizada sería necesario reemplazar el servicio en segundo plano por una implementación propia para las pruebas. Debido al mecanismo que emplea Android para el enlazado a los servicios, no se encontró la forma de reemplazar esta dependencia. Por lo tanto, las pruebas de integración se realizaron manualmente, utilizando la aplicación MetaWear Emulator desarrollada anteriormente para verificar el correcto funcionamiento de todos los casos de uso implementados.

Tras finalizar la iteración y haber realizado las pruebas, el cliente pudo valorar el prototipo resultante y sus críticas se tuvieron en cuenta para el desarrollo posterior.

5.4.2 Iteración 2

Esta iteración se ha centrado en el desarrollo de todo lo relativo a la gestión de las alertas, la detección de las distintas condiciones de alerta en base a los datos de los sensores y varias formas de avisar al usuario cuando estas se produzcan.

Análisis

Requisitos funcionales:

- Detección de convulsiones a partir de los datos de aceleración.

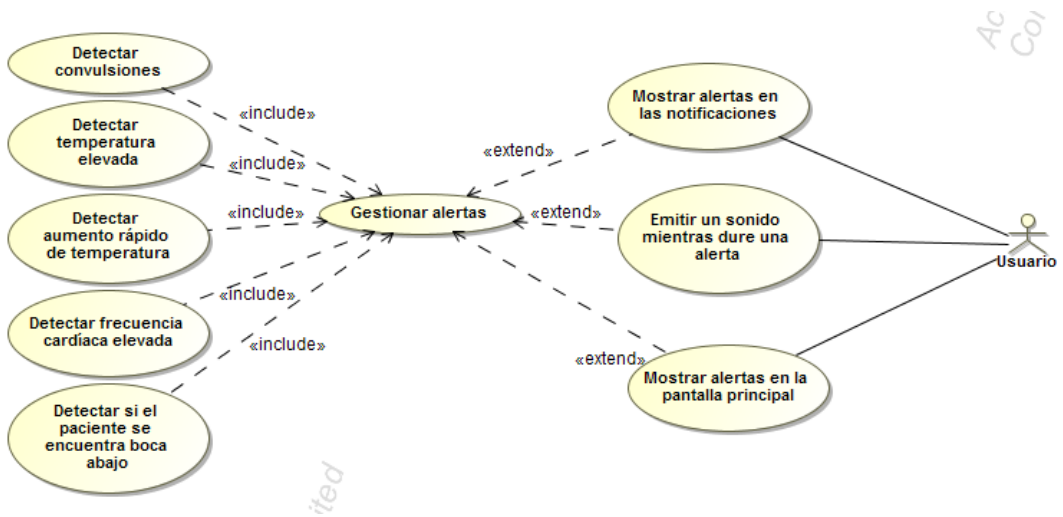


Figura 5.15: Casos de uso de la segunda iteración

- Detección de temperatura elevada.
- Detección de aumento repentino de temperatura.
- Detección de frecuencia cardíaca elevada.
- Detección de orientación boca abajo.
- Notificación de alertas.
- Reproducción de sonido de alerta.
- Mostrar alertas en la pantalla principal.

Requisitos no funcionales:

- Mientras esté conectado al wearable, alertar al usuario aunque no tenga la aplicación abierta.
- El algoritmo de detección debe ser lo suficientemente ligero para su ejecución en tiempo real.

Una vez establecida la conexión con el dispositivo wearable que lleva el paciente, los datos que producen los sensores se irán comprobando en todo momento para detectar alguna anomalía que requiera alertar al usuario. Estas comprobaciones se realizarán en el servicio en segundo plano, de modo que aunque el usuario cambie de aplicación o tenga el teléfono bloqueado pueda ser alertado igualmente.

Para la detección de convulsiones, se tratará de identificar movimientos repentinos que pudiesen corresponderse con los espasmos que pueden producirse en estos pacientes, y que se mantengan de forma periódica durante un tiempo determinado. Tanto para el caso de una temperatura elevada como el de la frecuencia cardíaca elevada basta con ir comprobando si el valor actual supera cierto umbral. También se comprobará si las últimas lecturas en un período corto de tiempo indican un aumento muy pronunciado de la temperatura. Finalmente, también podrá comprobarse si la orientación del dispositivo wearable se invierte, para aquellos cuidadores que quieran saber si el paciente está boca abajo. En la siguiente iteración se incluirá la posibilidad ajustar varios parámetros que determinan la sensibilidad de detección de las alertas.

Cuando se produzcan una o más alertas, el usuario puede recibir esta información de varias maneras. Por un lado, en la pantalla principal se mostrará un mensaje de estado, que en caso de producirse alguna alerta cambiará de color. En caso de producirse varias alertas simultáneamente, dicho mensaje informará únicamente de la más grave. El mismo mensaje de estado se mostrará también en la notificación permanente que requiere Android para ejecutar un servicio en segundo plano. Además, según se vayan produciendo alertas, se irán añadiendo a una lista que se muestra en una segunda notificación. Cada nueva alerta también provocará la vibración del dispositivo. Por último, se reproducirá un sonido de alerta mientras se esté produciendo alguna alerta.

Diseño

La detección de las condiciones de alerta, se realizó en cuatro clases separadas, una para cada tipo de datos que recibimos de la placa. `MonitorService` se encarga de ir pasando los datos a la clase que corresponda. Para representar cada una de las alertas, se empleó la clase `Alert`. Se empleó un tipo enumerado para identificar las distintas alertas que se pueden producir. Para determinar qué alerta tiene mayor prioridad se utiliza el orden en el que se definan los valores del enumerado.

Cada una de estas clases se comunica con `AlertManager` para indicarle cuándo comienza o termina una alerta. `AlertManager` se encarga de mantener una lista de las alertas que se han producido, una lista de las alertas que todavía no han terminado y cuál de estas últimas tiene mayor prioridad; siendo esta la que deberíamos mostrar en los mensajes de estado.

Para notificar a diferentes componentes de la aplicación del estado de las alertas, se utilizó el patrón observador. `AlertManager` desempeña el rol de observable, y se decidió implementar métodos para que los observadores puedan obtener el estado de este. De esta manera

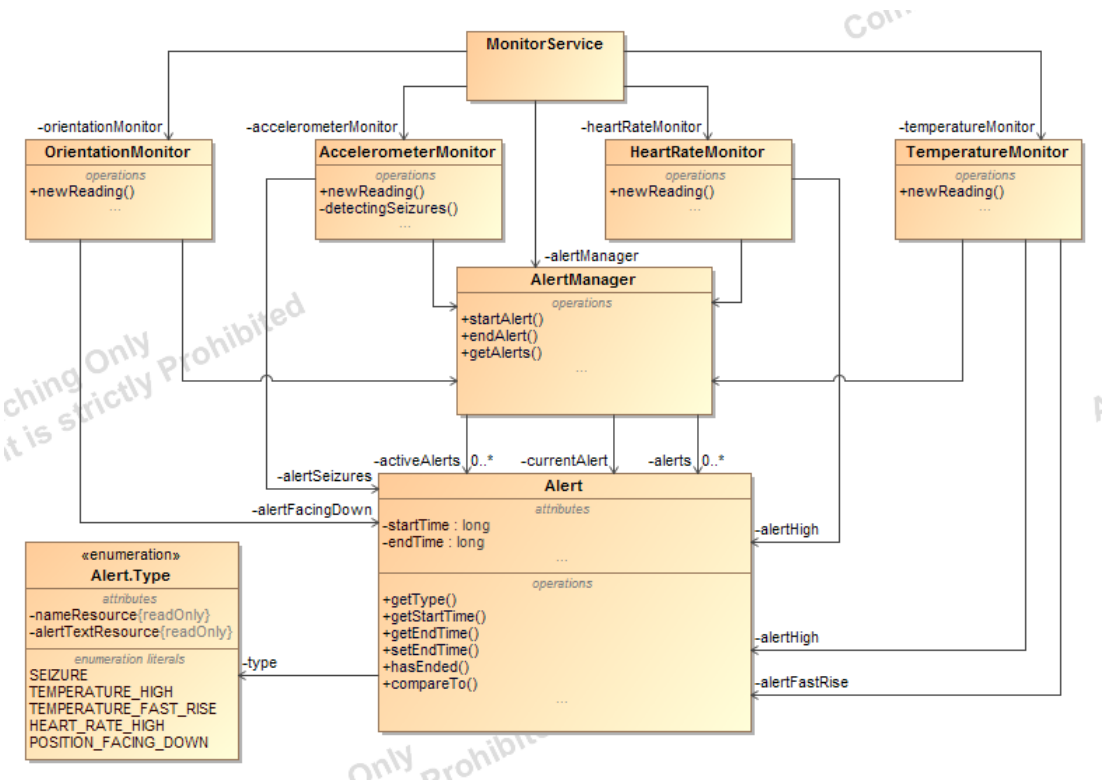


Figura 5.16: Diagrama de clases de la segunda iteración. Detección de alertas

podemos notificar dos eventos diferentes sin necesidad de dos observables. Siempre que se inicie o termine una alerta se notifica a los observadores. Adicionalmente, si esto implica que la alerta más prioritaria ha cambiado, incluyendo cuando se terminan todas las alertas o sólo hay una alerta en ese momento, se procede a notificarlos una segunda vez. A través del método `hasCurrentAlertChanged()` los observadores saben de qué evento se trata.

`LiveDataActivity`, `Notifications` y `AlarmSoundPlayer` ejercen el rol de observador. Para los dos últimos, `MonitorService` se encarga de añadirlos a la lista de observadores de `AlertManager`. Para el caso de `LiveDataActivity`, se amplió el `Binder` con métodos para que los componentes que se enlazan al servicio puedan añadir y borrar observadores.

En `LiveDataActivity` se añadió un componente de texto en la parte superior de la interfaz que muestra un mensaje de estado dependiendo de la alerta que se produzca en cada momento. Para esta función, solo le interesa el evento de cambio de alerta más prioritaria, ya que solo mostrará el mensaje que se corresponda con esta. En el caso de `AlarmSoundPlayer`, su comportamiento también depende únicamente del mismo evento, para iniciar el sonido cuando se esté produciendo alguna alarma y detenerlo cuando terminen todas.

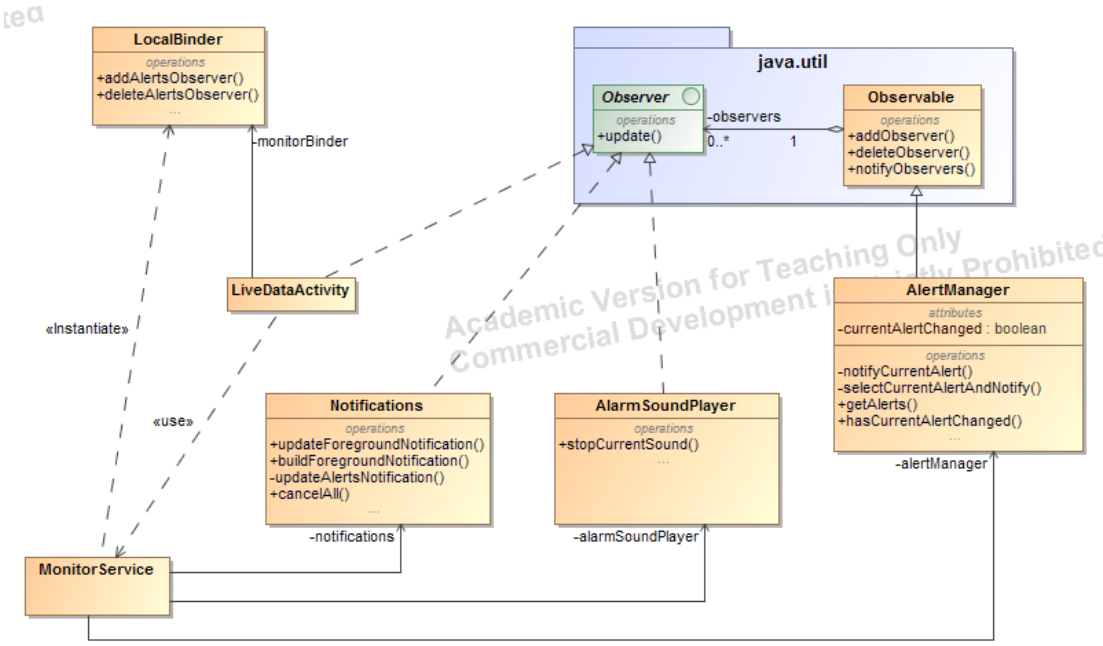


Figura 5.17: Diagrama de clases de la segunda iteración. Avisar al usuario de las alertas.

Al contrario que las dos anteriores, la clase `Notifications` sí reacciona a ambos tipos de eventos de `AlertManager`. Por un lado, se encarga de actualizar la notificación del servicio en primer plano con los mismos mensajes de estado que se muestran en `LiveDataActivity`. Por otro, muestra una segunda notificación con las últimas alertas que se han producido, mostrando la hora de inicio y su duración si han terminado.

En la última iteración se añadió comportamiento en respuesta al otro evento en `LiveDataActivity`, para implementar animaciones que resalten el inicio y fin de cada alerta.

Implementación

La detección de las convulsiones se implementó en la clase `AccelerometerMonitor`. En primer lugar, se calcula el módulo de la aceleración a partir de los componentes de los 3 ejes que envía el sensor. Para este caso de uso, el objetivo es encontrar movimientos bruscos, sin importar la dirección en la que se produzcan. Cuando el módulo de la aceleración supera un parámetro de umbral definido, se entiende que comienza un pico en la señal y se guarda una marca de tiempo. Cuando esta vuelve a estar por debajo del umbral el pico habrá terminado y volvemos a esperar un nuevo pico. Para iniciar una alerta, deben detectarse múltiples picos durante un tiempo prudencial para evitar falsos positivos dentro de lo posible. Además, si no se produce otro pico en un tiempo determinado, se considera que se detuvieron

las convulsiones. Esto se implementó utilizando un `Timer` para programar la detención de la alerta pasado ese tiempo. Solo si se siguen detectando estos movimientos bruscos continúa la alerta, cancelando el `Timer` e iniciando uno nuevo. En caso de que no hubiese pasado el tiempo necesario para comenzar la alerta, se reinicia el tiempo del primer pico para empezar de nuevo. En la siguiente iteración se implementó la modificación por parte del usuario de algunos parámetros de las alertas, como el umbral para las convulsiones y el tiempo necesario para que se inicie la alerta.

La posición que nos envía la placa puede tomar 8 valores diferentes. Cuatro de ellos indican que el dispositivo está boca abajo y los otros cuatro que está boca arriba. `OrientationMonitor` solo mantiene como estado un valor booleano para saber si el wearable se encuentra boca abajo o boca arriba. Como esto es lo único que se muestra en la interfaz, no se envían a la actividad todos los cambios de posición recibidos, sino que solo se notifica cuando pasa a estar boca abajo y viceversa. Al mismo tiempo, también se inicia la alerta o se detiene a través de `AlertManager`.

Para las alertas de temperatura elevada y frecuencia cardíaca elevada, el funcionamiento es el mismo. Cuando los valores leídos superan un umbral definido se inicia una alerta, y cuando descienden dicho umbral ésta se detiene. En `TemperatureMonitor`, además de esta alerta, también se implementó otra alerta para el caso en el que la temperatura, independientemente de que supere el umbral mencionado, se haya incrementado muy rápidamente. Para esta alerta, se guarda una lista con las últimas lecturas, y se va comprobando si el incremento de temperatura es mayor del deseado para ese período. En caso de que la temperatura disminuya, se descartan los valores anteriores que sean mayores a la última lectura.

Para la actualización del mensaje de estado en `LiveDataActivity` se implementó el interfaz `Observer`, como se describió en el apartado anterior. Un detalle importante es que al utilizar el patrón observador, el código del método `update()` de la actividad se ejecutará en el mismo hilo en el que se ejecute el observable. En este caso, ese hilo no es el que Android llama como principal o de interfaz de usuario, lo que significa que no se puede acceder directamente a los componentes del interfaz. Para realizar cualquier operación sobre los elementos gráficos, se utilizó el método `runOnUiThread()`. Esto no es necesario donde utilizamos un `BroadcastReceiver`, ya que este se encarga de que la ejecución en su método `onReceive()` sea siempre en el hilo de UI.

En `Notifications` se implementó todo lo relativo a las notificaciones de Android. Ahora la notificación permanente del servicio se construye en esta clase, en lugar de en `MonitorService`, y se ofrecen métodos para que el servicio haga uso de esta notificación. Por lo general, el mensaje mostrado se actualiza en función de la alerta activa con mayor prioridad, si es que se está produciendo alguna. También puede actualizarla el servicio en algunos

casos, como por ejemplo, para indicar que se está intentando reconectar tras haber perdido la conexión. En cuanto a la segunda notificación, en su título se muestra el número de alertas que se han producido hasta el momento y en su cuerpo se muestran las últimas 6, una alerta por línea. No se pueden mostrar más de 6 líneas en este tipo de notificaciones. Por cada alerta, se muestra la hora de inicio, su nombre y en caso de que haya terminado, su duración. Para una visualización más clara, se diferenció el estilo del texto de las alertas que todavía se están produciendo.

Para la reproducción del sonido de alerta, se utilizó la clase `MediaPlayer` del paquete `android.media`. `AlarmSoundPlayer` implementa el interfaz `Observer` y gestiona el reproductor según el estado de las alarmas. Cuando se produce alguna alerta, se configura el `MediaPlayer` con el sonido escogido, indicando que se reproduzca en bucle, y se inicia la reproducción. Para esta iteración, el sonido utilizado es el que esté definido en el sistema como sonido por defecto para las alarmas. Si la alerta con mayor prioridad cambia, se reinicia la reproducción desde el inicio del sonido. El sonido durará hasta que todas las alertas terminen, momento en el que se detiene el reproductor y se liberan los recursos utilizados. El volumen puede ajustarse con los controles de volumen del sistema, con el correspondiente al volumen de las alarmas.

Pruebas

Durante esta iteración, se implementaron pruebas de unidad para verificar que los componentes que se encargan de la detección de las alertas y su gestión se comportan según la especificación deseada. Las bibliotecas utilizadas son las mismas que en las iteraciones anteriores.

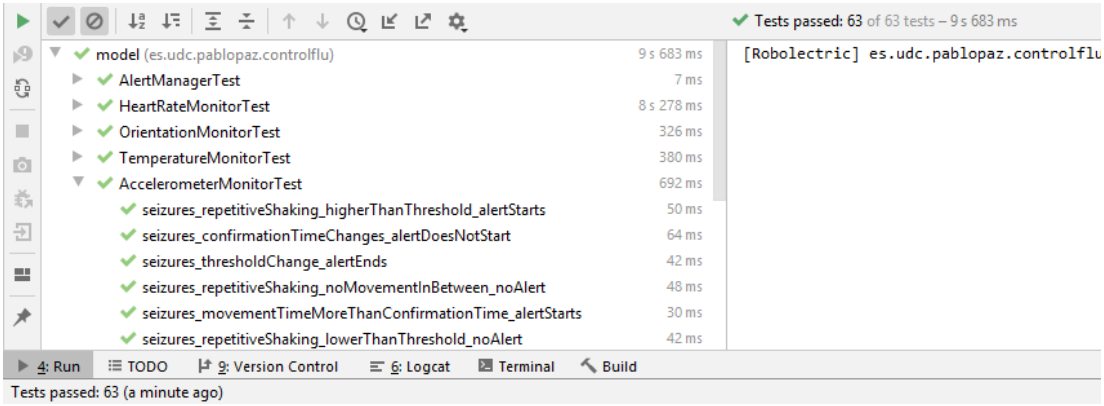


Figura 5.18: Pruebas unitarias. Detección de alertas

Del mismo modo que en la iteración anterior, las pruebas de integración se realizaron de

forma manual, comprobando todas las vías de ejecución para los casos de uso implementados.

5.4.3 Iteración 3

En esta iteración implementaremos las gráficas que permitirán al usuario monitorizar fácilmente y en tiempo real, el estado del paciente. También añadiremos una pantalla de ajustes y una pantalla que muestre los valores máximos y mínimos de la última semana, guardando en una base de datos las lecturas que correspondan.

Análisis

Requisitos funcionales:

- Mostrar gráficas en tiempo real con los datos recibidos de los sensores.
- Seleccionar qué gráficas queremos ver.
- Permitir el ajuste de algunos parámetros de detección de las alertas.
- Permitir que el usuario pueda seleccionar el sonido para las alertas.
- Guardar los datos de temperatura y frecuencia cardíaca en una base de datos.
- Visualización en forma de gráficas de los valores máximos y mínimos diarios de la última semana.
- Mejoras de la interfaz y retoques finales del aspecto de la aplicación.

Requisitos no funcionales:

- Información actualizada en tiempo real, el usuario no debe apreciar una latencia significativa.
- Internacionalización de la aplicación.

El usuario podrá ver en la pantalla principal gráficas que se actualizan en tiempo real para los datos de aceleración, frecuencia cardíaca y temperatura. El objetivo es que el usuario pueda obtener de un vistazo toda la información relevante, reservando la mayoría del espacio a una visualización cómoda de las gráficas. Una vez establecida la conexión, el usuario podrá ver como van apareciendo los datos hasta llenar la pantalla. También podrá realizar algunos

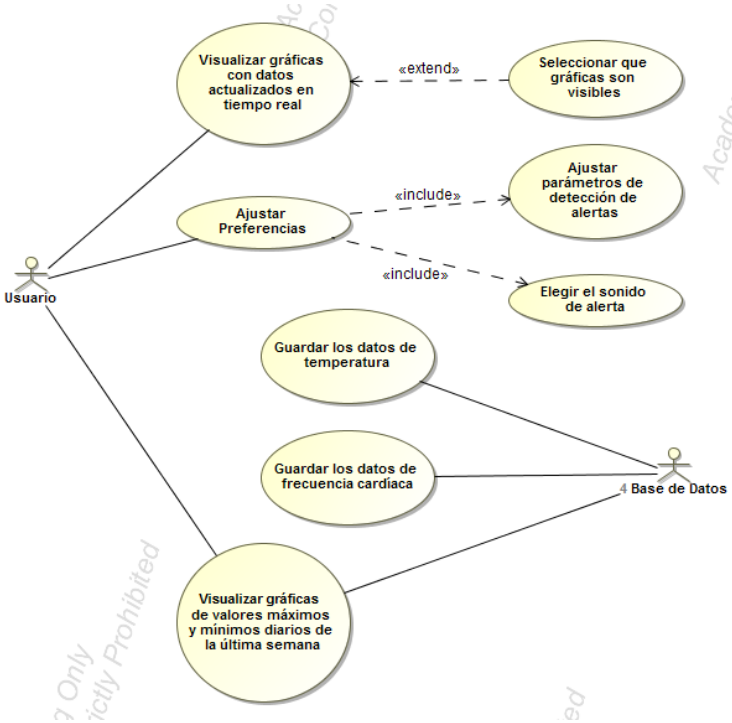


Figura 5.19: Casos de uso de la tercera iteración

gestos táctiles sobre la superficie de estas, para hacer scroll o cambiar la escala de los datos. Si el usuario pulsa un botón situado en la zona superior de la pantalla, se desplegará un menú que le permitirá seleccionar o deseleccionar las gráficas que serán visibles. Las gráficas siempre ocuparán todo el espacio de pantalla disponible, por lo que el usuario podrá escoger ver una sola gráfica en grande, las tres más pequeñas o la combinación que desee.

Se habilitará otro botón en la barra de menú para abrir una pantalla de ajustes. Una vez en esta pantalla, se le presentarán al usuario la posibilidad de ajustar diversos parámetros que influirán en la detección de las alarmas. Se podrán modificar los umbrales de frecuencia cardíaca y temperatura a partir de los cuales queremos que se nos alerte. Para la detección de convulsiones, podremos modificar cómo de intensos deben de ser los movimientos que se detectarán, así como el tiempo que se deben de mantener de forma periódica para que se lance la alerta. El usuario también podrá activar o desactivar las alertas por la orientación del paciente. Otra opción que se habilitará en esta misma pantalla, permitirá que el usuario seleccione, entre los disponibles en su dispositivo, el sonido de alarma que desee.

Un tercer botón en la barra superior permitirá al usuario navegar a una nueva pantalla, que le ofrecerá información de la última semana. En esta pantalla podrá visualizar dos gráficas con los valores máximos y mínimos diarios registrados para la temperatura y frecuencia cardíaca.

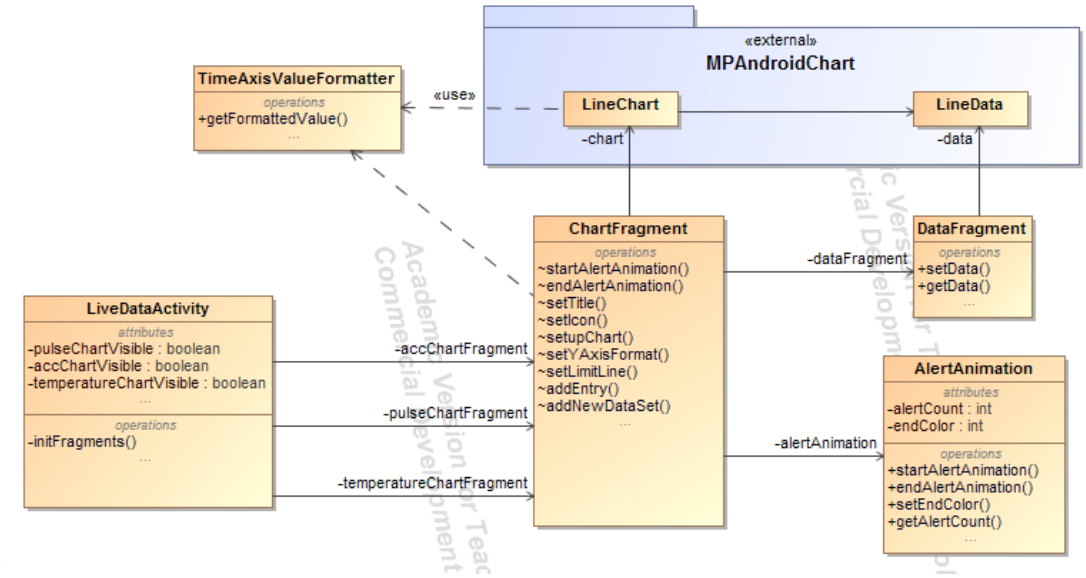


Figura 5.20: Iteración 3. Diagrama de clases. Gráficas en tiempo real

Diseño

Para todas las gráficas que se muestran en la aplicación, se utilizó la biblioteca *MPAndroidChart*[40]. En cuanto a las gráficas que muestran los datos en tiempo real, todo el comportamiento de la gráfica y el uso de la mencionada biblioteca se encapsula en un fragment que permita ajustar ciertos parámetros. De este modo, se reutiliza para los datos de los tres sensores que queremos mostrar. De esta forma, también sería muy sencillo añadir gráficas para nuevos sensores en el futuro. El fragment `ChartFragment` hace uso de `LineChart`, uno de los tipos de gráficas de la biblioteca.

Para el caso de uso de los valores máximos y mínimos de la última semana, necesitamos implementar persistencia para la temperatura y la frecuencia cardíaca. Siguiendo las recomendaciones de la documentación de Android, se decidió utilizar la biblioteca *ROOM*, que brinda una capa de abstracción sobre SQLite, para la persistencia. También se decidió utilizar otros componentes de las bibliotecas de arquitectura de Android como *Viewmodel* y *LiveData*. Esto permite seguir una arquitectura probada y extensible que dispone de buena integración con *ROOM*. La clase `Repository` realiza las operaciones de inserción y las consultas utilizando los DAOs. El resto de componentes de la aplicación utilizan el `Repository` para todo lo relacionado con la persistencia. Este diseño permitiría modificar únicamente la clase `Repository` si se quisiera añadir otras fuentes de datos, como por ejemplo para algún tipo de sincronización en la nube.

El componente *Viewmodel* está diseñado para almacenar datos de la interfaz, evitando que

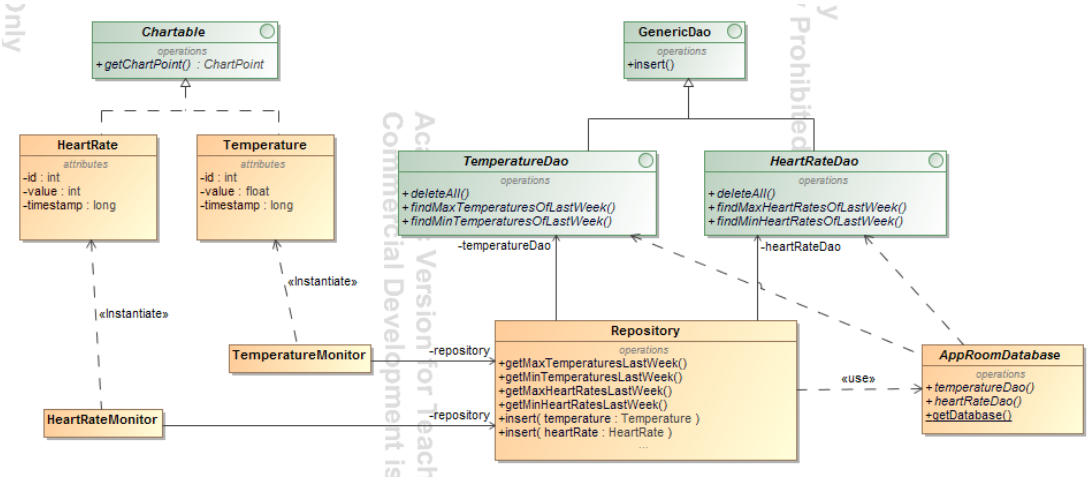


Figura 5.21: Iteración 3. Diagrama de clases. Persistencia

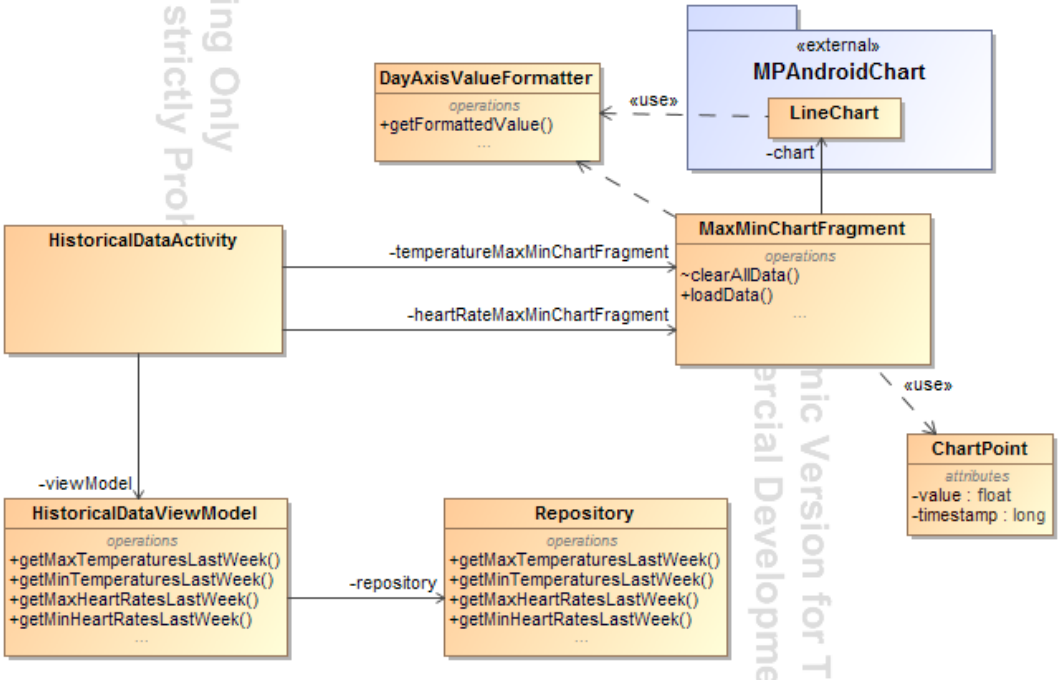


Figura 5.22: Iteración 3. Diagrama de clases. Gráficas de la última semana

la actividad tenga esta responsabilidad. Además, el *Viewmodel* sobrevive a los cambios de configuración, por lo que evita que tengamos que guardar y recuperar manualmente estos datos. En este caso, `HistoricalDataViewModel` se encarga de obtener los datos máximos y mínimos de la última semana para que `HistoricalDataActivity` pueda acceder a ellos siempre que lo necesite.

De forma similar a lo hecho para las gráficas de la pantalla principal, se creó un fragment que gestione una gráfica para los valores máximos y mínimos diarios de la última semana. `HistoricalDataActivity` contiene dos de estos fragments, uno para la temperatura y otro para la frecuencia cardíaca. Para que el fragment no dependiese de las entidades concretas, creamos una clase `ChartPoint`, que representa un punto a dibujar en la gráfica, e hicimos que las entidades implementasen el interfaz `Chartable`.

Implementación

En el archivo de layout de la actividad principal se añadieron tres `ChartFragment` de manera que se adapten al espacio disponible. Para poder ajustar programáticamente las gráficas a cada tipo de datos que queremos mostrar, se implementó en `ChartFragment` el método *setupChart(xRange, yMin, yMax, yGranularity, circleRadius, color)* que la actividad se encarga de llamar al inicio de su ciclo de vida. Excepto el rango para el eje *x*, para el resto de parámetros puede pasarse un valor nulo, adoptando una configuración automática o desactivando la característica correspondiente.

El rango del eje *x* define el rango de tiempo que visualizaremos en la gráfica. Por ejemplo, debido a la frecuencia con la que se reciben datos de aceleración, para esta gráfica se decidió ajustar este parámetro para ver tan sólo los últimos 10 segundos en tiempo real. Al desplazar la gráfica para ver datos anteriores, la escala del eje *x* se mantiene.

También se pueden ajustar los valores mínimos y máximos para el eje *y*; así como la separación de las etiquetas de este eje. Valores nulos para estos parámetros provocan que se utilicen los valores que la biblioteca calcula automáticamente como adecuados. Para las gráficas utilizadas se ajustaron los valores mínimos del eje *y*, dejando que los valores máximos se calculen automáticamente. De esta manera, si los valores de los sensores son mayores que el máximo del eje, la biblioteca lo cambia para que se puedan ver los nuevos datos. También se puede elegir si queremos mostrar un círculo por cada dato o no. Para la temperatura y la frecuencia cardíaca se muestran, mientras que para la aceleración se optó por mostrar los datos como una línea continua.

La biblioteca *MPAndroidChart* permite personalizar el formato de las etiquetas que se muestran en cada eje implementando el interfaz `IAxisValueFormatter`. Para el eje

y, la actividad pasa el formato adecuado al fragment y este implementa el interfaz como una clase anónima que utilice dicho formato. Para el eje *x*, se implementó el interfaz en la clase `TimeAxisValueFormater` que, para cada etiqueta que quiera imprimir la biblioteca, calcula cuanto tiempo hace desde que se recibió el dato en cuestión.

También se implementó un método que permite a la actividad añadir una línea horizontal discontinua para representar visualmente el umbral definido en la aplicación para las alertas. Finalmente, se puede personalizar el icono y el título que se muestran como encabezamiento de las gráficas.

Cabe recordar que `LiveDataActivity` recibe del servicio los datos de los sensores en un `BroadcastReceiver`. En este, ahora la actividad pasa cada valor y su timestamp, que el API de MetaWear nos devuelve conforme al momento en que se recibe el dato, al `ChartFragment` adecuado. En el fragment, se añade una nueva entrada a la gráfica y, en caso de que el usuario no esté desplazando la gráfica, se mueve la vista para que la gráfica se mueva con los nuevos datos recibidos.

Para no perder los datos añadidos a las gráficas ante los cambios de configuración de Android, que se producen cada vez que se cambia la orientación del teléfono, se optó por utilizar otro fragment para guardar el objeto que contiene los datos de la gráfica. Android nos permite establecer que un fragment no se recree cuando lo haga la actividad. Esto es lo que se hizo con `DataFragment`, que no tiene interfaz y cuyo único objetivo es mantener los datos en memoria y sobrevivir a la recreación de la actividad.

Se añadió un menú en la barra superior de la actividad principal, con tres elementos. Los dos elementos que sirven para navegar a otras actividades están escondidos hasta que el usuario pulse el icono de tres puntos verticales. El único elemento que está directamente en la barra en forma de icono es el que despliega un submenú que permite al usuario seleccionar o deseleccionar las gráficas que quiere ver. Al deseleccionar una gráfica, se oculta el fragment de dicha gráfica, ocupando las restantes el espacio disponible. Al contrario, cuando se selecciona, vuelve a mostrarse la gráfica repartiendo el espacio entre ellas. La visibilidad de cada gráfica se guarda en las preferencias de la aplicación, para que cuando el usuario vuelva a utilizar la aplicación se encuentre la misma disposición que había seleccionado la última vez.

Para que el usuario pueda ajustar ciertos parámetros de la detección de alertas se añadió una pantalla de ajustes. Se utilizó la biblioteca *Preference* de la plataforma, que ofrece una experiencia consistente para los usuarios de Android y se encarga de almacenar pares clave-valor en las preferencias de la aplicación. Se definió la jerarquía de las preferencias en un archivo *xml* y estas se añaden en un `PreferenceFragment`.

La preferencia que permite activar o desactivar la alerta de que el paciente está boca abajo

se implementó utilizando `SwitchPreference` y para la selección del sonido de las alertas se utilizó `RingtonePreference`, ambas del paquete `android.preference`. Para el resto se implementaron tres preferencias personalizadas, que abren un diálogo en el que el usuario puede escoger el valor deseado en un selector numérico con un rango predefinido. Para desarrollar `NumberPickerPreference` se tomó como base la siguiente implementación [41], adecuada para seleccionar valores enteros, por lo que se utilizó para el umbral de frecuencia cardíaca y el tiempo de confirmación para las convulsiones. Se desarrollaron dos modificaciones para poder seleccionar valores decimales: `DecimalNumberPicker`, que utiliza dos selectores para la parte entera y la decimal y que se empleó para ajustar el umbral de temperatura; y `FloatNumberPicker`, que añade un parámetro de configuración adicional que ajusta el incremento de los valores del selector y que se utilizó para la preferencia del umbral de aceleración. Finalmente, para que los ajustes tengan efecto, se hizo que las clases que detectan las alertas obtengan los parámetros a partir de las preferencias antes de cada comprobación. También se modificó `AlarmSoundPlayer` para establecer el sonido seleccionado antes de iniciar la reproducción.

Se incluyeron las dependencias de compilación necesarias para utilizar *ROOM* y el resto de componentes de arquitectura de Android. *ROOM* proporciona una serie de anotaciones que nos sirven para describir las entidades y DAOs, y a partir de estas genera el código necesario para la implementación de la persistencia utilizando *SQLite*. Se anotaron las dos entidades, utilizando un identificador generado automáticamente como clave primaria.

En cuanto a los DAOs, se definen como interfaces o clases abstractas y *ROOM* genera la implementación concreta según las anotaciones. Ofrece anotaciones para las operaciones más comunes; como inserción, actualización y borrado. Se decidió definir un DAO genérico en el que se declara un método para la inserción, común a todas las entidades, con la anotación por defecto para la inserción. En los DAOs para cada entidad se definen las consultas SQL para obtener los máximos y mínimos diarios de la última semana y el borrado de todas las filas de la tabla.

Se implementó el patrón instancia única para ofrecer un único punto de acceso al componente de la base de datos, `AppRoomDatabase`. Los métodos de los DAOs con los que accedemos a la base de datos, deben ejecutarse en un hilo que no sea el principal, para no degradar el rendimiento de los componentes de la interfaz. En `Repository`, se implementó una clase que hereda de `AsyncTask` para realizar las inserciones en otro hilo de ejecución de manera asíncrona. Para las consultas, *ROOM* puede devolver los resultados dentro de un objeto `LiveData`, de las bibliotecas de arquitectura. Este es un objeto observable que *ROOM* actualiza ejecutando de nuevo la consulta en cuanto se produzcan cambios en las tablas de la base de datos. Además, se encarga automáticamente de ejecutar las consultas en un hilo en

segundo plano. Tanto `Repository` como `HistoricalDataViewModel` devuelven este `LiveData` para que `HistoricalDataActivity` pueda supervisar los cambios y disponer siempre de los datos actualizados.

A modo de ejemplo, aquí se muestra la consulta, utilizando la anotación @Query de ROOM, para obtener los valores máximos de temperatura diarios de la última semana. El resto de consultas son muy similares.

```
@Query("SELECT id, max(value) as value, timestamp " +
       "FROM (" +
       "   SELECT * FROM temperature " +
       "   WHERE timestamp >= strftime('%s', 'now', '-6 days',
    'start of day')*1000" +   // Get data from last week
       ") " +
       "GROUP BY date(timestamp/1000, 'unixepoch') " +//Group by day
       "ORDER BY timestamp ASC ")
LiveData<List<Temperature>> findMaxTemperaturesOfLastWeek();
```

Tal y como se hizo para las gráficas en tiempo real, la personalización y comportamiento de las gráficas de la última semana se encapsuló en un fragment, `MaxMinChartFragment`. La actividad le pasa la lista de valores máximos y la de valores mínimos por separado, ya que estos se dibujan con tonos de color diferentes y se representan unidos por líneas para visualizar las tendencias. También se implementó un formato personalizado para las etiquetas del eje *x*, en la clase `DayAxisValueFormatter`, para mostrar los días de la semana. Al utilizar `LiveData` para recibir los resultados de las consultas, si durante la visualización de estas gráficas se reciben datos de los sensores mayores que el máximo o inferiores al mínimo para ese día, las gráficas se actualizan para reflejarlo.

Para finalizar, se llevaron a cabo los retoques finales del aspecto de la aplicación, añadiendo animaciones en la pantalla de datos en tiempo real. De esta forma, mientras se produzca alguna alerta, se resaltan la gráfica y el dato en la barra inferior relevantes a esta, llamando la atención del usuario. También se llevó a cabo la internacionalización de todos los textos que aparecen en la interfaz de usuario a tres idiomas: inglés, gallego y castellano. Para ello, todos los textos se externalizaron a un archivo de recursos *strings.xml* para cada código de idioma.

Pruebas

En cuanto a las pruebas de unidad, además de pruebas para componentes añadidos en esta iteración, se añadieron casos de prueba a las clases implementadas en la iteración anterior para verificar que el ajuste de los parámetros de las alertas produce el comportamiento correcto. Tal y como recomienda la documentación, las pruebas que utilizan la base de datos se realizaron

para ser ejecutadas en un dispositivo real. Esto es recomendable para que las pruebas utilicen la versión de Sqlite disponible en el dispositivo en lugar de el de la máquina de desarrollo. Además, al no depender de componentes de interfaz, se ejecutan razonablemente rápido. Para que las pruebas sean más herméticas, se utilizó una versión en memoria de la base de datos.

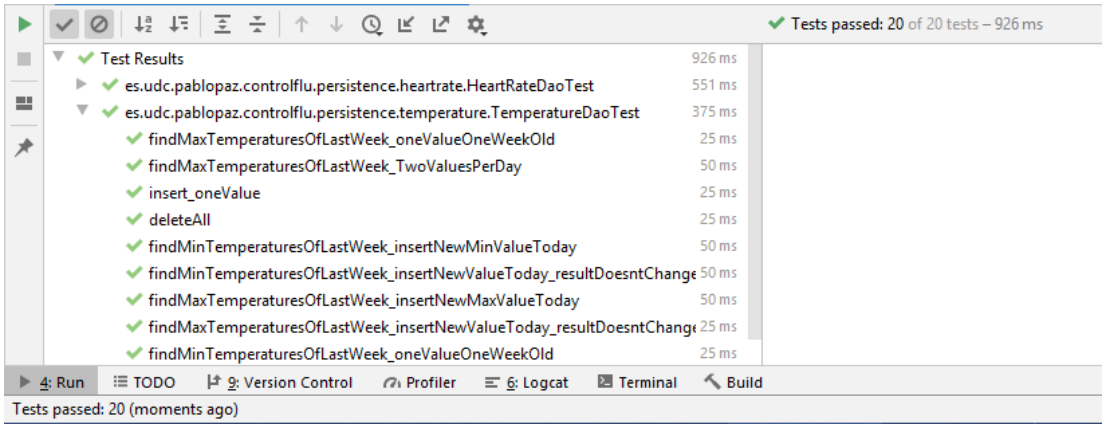


Figura 5.23: Pruebas de unidad de los DAOs

Finalmente, utilizando la aplicación MetaWear Emulator, se llevaron a cabo pruebas manuales del sistema en su conjunto y de todos los caso de uso desarrollados, comprobando la validez de la implementación realizada.

Capítulo 6

Resultados

En líneas generales, se han conseguido satisfacer los objetivos que se habían establecido para ambas aplicaciones.

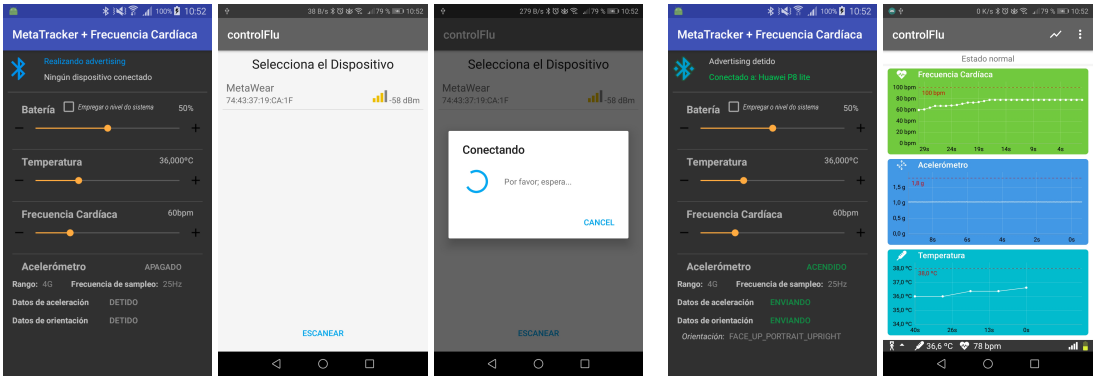


Figura 6.1: Conexión entre ambas aplicaciones desarrolladas

En cuanto a la aplicación de emulación, el resultado obtenido ha sido el esperado, lo que ha permitido utilizarla para el desarrollo de la aplicación ControlFlu. Ante la falta de experiencia con el sistema Android y la tecnología BLE, no se podía estar seguro de la viabilidad de implementar la comunicación ejerciendo el rol de periférico y de servidor GATT, fundamental para el funcionamiento de la aplicación. Finalmente, el funcionamiento conseguido a este respecto es satisfactorio, y de igual manera el control de los datos producidos y enviados ha sido el deseado. Un punto a considerar es que con los sensores emulados y la relativa similitud entre los modelos de este fabricante actualmente disponibles, la funcionalidad de selección de modelo no ofrece un gran valor a la aplicación, más allá de seleccionar la configuración con un sensor adicional para la frecuencia cardíaca. Sin embargo, se considera que el diseño ha cumplido el objetivo de permitir la extensión futura, y una implementación del resto de sensores y módulos así como la posibilidad de añadir nuevos modelos de placa añadirían valor a

esta característica.

Figura 6.2: Monitorización con ControlFlu. Datos producidos por MetaWear Emulator

La aplicación ControlFlu ha cumplido con las expectativas, permitiendo la monitorización en tiempo real de las constantes vitales de un paciente pediátrico y alertando en caso de valores anormales de estas o posibles episodios convulsivos. Dicho esto, para la comercialización de una solución de monitorización a partir de este desarrollo, sería necesario comprobar la precisión de los valores medidos por los sensores en comparación con equipamiento especializado, así como el rendimiento del algoritmo utilizado para la detección de convulsiones y la proporción de falsos positivos en relación a los enfoques utilizados clínicamente. Por ejemplo, para el desarrollo de nuestra aplicación se asumió que la medición de temperatura que proporcione el wearable con su sensor integrado sería útil para detectar aumentos de la temperatura corporal. Sin embargo, podría resultar necesario incluir un sensor de temperatura externo en el producto final para conseguir un resultado satisfactorio o modificar el tipo de información mostrada dependiendo de su fiabilidad. También sería necesario determinar la colocación idónea del wearable en el paciente, tal que el acelerómetro pueda recoger los movimientos convulsivos adecuadamente. En conclusión, este proyecto podría servir como base para una solución de monitorización a partir de un nuevo modelo de placa más enfocado en la salud, o un desarrollo que añada algún sensor necesario a partir de los modelos existentes.

Capítulo 7

Conclusiones

En los últimos años, han proliferado comercialmente multitud de dispositivos inalámbricos con un factor de forma pequeño y que incluyen diversos sensores con el objetivo de recoger datos. En parte, esto se debe al desarrollo de la tecnología BLE, ya que debido a su eficiencia energética ha aumentado la utilidad y viabilidad de este tipo de dispositivos. Podemos encontrar pulseras para monitorizar la actividad física, smartwaches con sensores de pulso y una gran variedad de dispositivos que se podrían englobar en lo llamado como Internet of Things. Con la realización de este proyecto se ha podido comprobar que estas tecnologías están lo suficientemente maduras para desarrollar productos de consumo que puedan mitigar algunos problemas en el ámbito de la salud. Todavía no se encuentran muchas soluciones en este ámbito, por lo que se presenta como una oportunidad de negocio interesante y que posiblemente experimente un gran crecimiento en los próximos años.

Con dispositivos como los que fabrica MbientLab, es posible desarrollar productos de monitorización útiles y cómodos sin una gran complejidad de desarrollo y a un coste competitivo. Cabe esperar que en un futuro cercano mejore la variedad de modelos y sensores, lo que facilitaría obtener del fabricante un producto que se adapte mejor al caso de uso.

La realización de este proyecto ha servido para afianzar los conocimientos adquiridos durante el Grado, además de para conocer y aprender a utilizar nuevas tecnologías. Antes del desarrollo de este trabajo, no se disponía apenas de experiencia en el desarrollo de aplicaciones móviles. Por lo tanto, la realización de este proyecto ha supuesto un aprendizaje de las tecnologías necesarias para el desarrollo de aplicaciones nativas para Android y del ecosistema que lo rodea. También se ha podido descubrir la tecnología de comunicación inalámbrica Bluetooth Low Energy, así como profundizar en ciertos detalles de su funcionamiento. Se ha aprendido a utilizar esta tecnología de comunicación en el sistema Android tanto en el rol central como en el de periférico. Además, ha servido como un acercamiento a los dispositivos

wearables y sus tecnologías y, más concretamente, a los dispositivos fabricados por Mbientlab. Finalmente, y de forma general, la realización de este trabajo ha redundado en una mayor experiencia personal en las diversas disciplinas involucradas en el desarrollo de software.

Capítulo 8

Futuros desarrollos

El ámbito de los dispositivos wearables está en constante evolución, por lo que para mantener o aumentar el valor aportado, las aplicaciones aquí desarrolladas deberían actualizarse acorde a dicha evolución. Existe la posibilidad de que el API MetaWear sufra cambios importantes en el futuro o MbientLab saque al mercado nuevos modelos de dispositivos wearables, caso en el que la actualización de ambas aplicaciones supondría el desarrollo futuro más importante.

Se han considerado varias posibles líneas futuras de desarrollo para la aplicación MetaWearEmulator:

- Implementación completa de las funcionalidades ofrecidas por el API MetaWear.
- Soporte para automatización de pruebas en aplicaciones cliente.
- Añadir emulación de algún otro tipo de dispositivo que se comunique a través de BLE.
- Opción de añadir o eliminar sensores de forma dinámica.

Como posibles desarrollos futuros para la app ControlFlu destacan:

- Refinamiento de la detección de alarmas en base a pruebas de uso reales y valoraciones de profesionales médicos.
- Añadir la captura y monitorización de algún sensor adicional que fuese compatible con las conexiones disponibles en las placas de MbientLab y que resultase útil para la detección precoz de convulsiones febriles. Se ha considerado interesante para este propósito un sensor de respuesta galvánica de la piel(GSR).

- Añadir más modos de visualizar datos históricos y alarmas registradas.
- Utilización de machine learning para tratar de anticipar las convulsiones[42, 43].
- Exportar datos de sensores o integración con algún servicio en la nube.
- Portar la aplicación al sistema iOS de Apple.

Bibliografía

[1] E. Pavlidou, C. Hagel, and C. Panteliadis, "Febrile seizures: Recent developments and unanswered questions," *Childs Nerv. Syst.*, vol. 29, no. 11, pp. 2011–2017, Nov. 2013, wOS:000326283900011.

[2] Commission on Epidemiology and Prognosis, International League Against Epilepsy, "Guidelines for Epidemiologic Studies on Epilepsy," *Epilepsia*, vol. 34, no. 4, pp. 592–596, 1993.

[3] S. Shinnar and T. Glauser, "Febrile seizures," *J. Child Neurol.*, vol. 17, no. SUPPL. 1, pp. S44–S52, 2002.

[4] A. Van De Vel *et al.*, "Non-EEG seizure-detection systems and potential SUDEP prevention: State of the art," *Seizure*, vol. 22, no. 5, pp. 345–355, 2013.

[5] S. Kusmakar, C. K. Karmakar, B. Yan, T. J.O'Brien, R. Muthuganapathy, and M. Palaniswami, "Automated Detection of Convulsive Seizures Using a Wearable Accelerometer Device," *IEEE Trans. Biomed. Eng.*, vol. 66, no. 2, pp. 421–432, Feb. 2019.

[6] A. A. Kabanov and A. I. Shchelkanov, "Development of a Wearable Inertial System for Motor Epileptic Seizure Detection," in *2018 XIV International Scientific-Technical Conference on Actual Problems of Electronics Instrument Engineering (APEIE)*, Oct. 2018, pp. 339–342.

[7] A. Dalton *et al.*, "Development of a Body Sensor Network to Detect Motor Patterns of Epileptic Seizures," *IEEE Trans. Biomed. Eng.*, vol. 59, no. 11, pp. 3204–3211, Nov. 2012.

[8] J. Gubbi, S. Kusmakar, A. S. Rao, B. Yan, T. O'Brien, and M. Palaniswami, "Automatic Detection and Classification of Convulsive Psychogenic Nonepileptic Seizures Using a Wearable Device," *IEEE J. Biomed. Health Inform.*, vol. 20, no. 4, pp. 1061–1072, Jul. 2016.

[9] I. Osorio and B. Manly, "Probability of detection of clinical seizures using heart rate changes," *Seizure*, vol. 30, pp. 120–123, 2015.

[10] "Gartner Says Huawei Secured No. 2 Worldwide Smartphone Vendor Spot, Surpassing Apple in Second Quarter 2018," [Accedido: 19-10-2019]. [En línea]. Disponible en: https://gtnr.it/31jrwQi

[11] "Android Jetpack," [Accedido: 18-01-2020]. [En línea]. Disponible en: https://developer.android.com/jetpack

[12] MbientLab, "MetaBase," [Accedido: 28-04-2020]. [En línea]. Disponible en: https://play.google.com/store/apps/details?id=com.mbientlab.metawear.metabase

[13] ——, "MetaWear Sample Android App," [Accedido: 22-05-2020]. [En línea]. Disponible en: https://github.com/mbientlab/MetaWear-SampleApp-Android

[14] Bluetooth Special Interest Group, "Bluetooth Core Specification 5.1," Jan. 2019.

[15] "Bluetooth Radio Versions," [Accedido: 17-11-2019]. [En línea]. Disponible en: https://www.bluetooth.com/bluetooth-technology/radio-versions/

[16] S. E. Stahl, H.-S. An, D. M. Dinkel, J. M. Noble, and J.-M. Lee, "How accurate are the wrist-based heart rate monitors during walking and running activities? Are they accurate enough?" *BMJ Open Sport Exerc. Med.*, vol. 2, no. 1, p. e000106, Apr. 2016.

[17] T. Choksatchawathi *et al.*, "Improving Heart Rate Estimation on Consumer Grade Wrist-Worn Device Using Post-Calibration Approach," *IEEE Sens. J.*, vol. 20, no. 13, pp. 7433–7446, 2020.

[18] MbientLab, "MetaHealth," [Accedido: 14-10-2019]. [En línea]. Disponible en: https://www.kickstarter.com/projects/guardyen/hr-gsr-motion-dev-board-for-health-and-fitness-pro

[19] movisens GmbH, "ECG and Activity Sensor - EcgMove 4," [Accedido: 04-10-2019]. [En línea]. Disponible en: https://www.movisens.com/en/products/ecg-sensor/

[20] Maxim Integrated, "MAXREFDES101#: Health Sensor Platform 2.0," [Accedido: 05-10-2019]. [En línea]. Disponible en: https://www.maximintegrated.com/en/design/reference-design-center/system-board/6779.html

[21] Polyglot Programming Inc, "Cordova plugin for the MetaWear API," [Accedido: 07-10-2019]. [En línea]. Disponible en: https://github.com/polyglotprogramminginc/cordova-plugin-metawear

[22] WebBluetoothCG, "BLE Peripheral Simulator," [Accedido: 14-10-2019]. [En línea]. Disponible en: https://play.google.com/store/apps/details?id=io.github.webbluetoothcg.bletestperipheral

[23] Nordic Semiconductor, "nRF Connect for Mobile," [Accedido: 07-01-2020]. [En línea]. Disponible en: https://play.google.com/store/apps/details?id=no.nordicsemi.android.mcp

[24] S. Tedesco, M. Sica, A. Ancillao, S. Timmons, J. Barton, and B. O'Flynn, "Accuracy of consumer-level and research-grade activity trackers in ambulatory settings in older adults," *PLOS ONE*, vol. 14, no. 5, p. e0216891, May 2019.

[25] S. Benedetto, C. Caldato, E. Bazzan, D. C. Greenwood, V. Pensabene, and P. Actis, "Assessment of the Fitbit Charge 2 for monitoring heart rate," *PLOS ONE*, vol. 13, no. 2, p. e0192691, Feb. 2018.

[26] M. P. Wallen, S. R. Gomersall, S. E. Keating, U. Wisløff, and J. S. Coombes, "Accuracy of Heart Rate Watches: Implications for Weight Management," *PLOS ONE*, vol. 11, no. 5, p. e0154420, May 2016.

[27] "NEEBO® - Child Well-Being Monitor," [Accedido: 10-05-2020]. [En línea]. Disponible en: https://neebomonitor.com/

[28] A. Cockburn, "Using both incremental and iterative development," *CrossTalk*, vol. 21, no. 5, pp. 27–30, 2008.

[29] MbientLab, "Tutorial Freefall," [Accedido: 10-11-2019]. [En línea]. Disponible en: https://mbientlab.com/tutorials/Java.html#freefall-app

[30] "Android 5.0 API," [Accedido: 24-01-2020]. [En línea]. Disponible en: https://developer.android.com/about/versions/android-5.0?hl=es

[31] Bluetooth Special Interest Group, "Developer Study Guide: Bluetooth Low Energy Security," Oct. 2019, [Accedido: 01-06-2020]. [En línea]. Disponible en: https://www.bluetooth.com/bluetooth-resources/le-security-study-guide/

[32] MbientLab, "MAC Unique Identifier," [Accedido: 29-04-2020]. [En línea]. Disponible en: https://mbientlab.com/tutorials/MACs.html

[33] Bluetooth Special Interest Group, "GATT Specifications," [Accedido: 12-06-2020]. [En línea]. Disponible en: https://www.bluetooth.com/specifications/gatt/

[34] K. Townsend, R. Davidson, Akiba, and C. Cufí, "Chapter 1. Introduction," in *Getting Started with Bluetooth Low Energy: Tools and Techniques for Low-Power Networking*, revised first edition ed. Sebastopol, CA: O'Reilly, 2014, oCLC: ocn870896083.

[35] MbientLab, "MetaWear Android API," [Accedido: 04-03-2020]. [En línea]. Disponible en: https://github.com/mbientlab/MetaWear-SDK-Android

[36] "FAQ – MbientLab," [Accedido: 20-05-2020]. [En línea]. Disponible en: https://mbientlab.com/faq/

[37] MbientLab, "Accelerometer — MetaWear SDK Android 3.7.0 documentation," [Accedido: 15-05-2020]. [En línea]. Disponible en: https://mbientlab.com/androiddocs/latest/accelerometer.html#global-enable

[38] MbientLab, "Temperature — MetaWear SDK Android 3.7.0 documentation," [Accedido: 16-05-2020]. [En línea]. Disponible en: https://mbientlab.com/androiddocs/latest/temperature.html

[39] "BoltsFramework/Bolts-Android," [Accedido: 02-03-2020]. [En línea]. Disponible en: https://github.com/BoltsFramework/Bolts-Android

[40] P. Jahoda, "MPAndroidChart," [Accedido: 26-05-2020]. [En línea]. Disponible en: https://github.com/PhilJay/MPAndroidChart

[41] Alobar Productions, "AndroidPreferenceTest," [Accedido: 12-04-2020]. [En línea]. Disponible en: https://github.com/Alobar/AndroidPreferenceTest

[42] G. Giannakakis, M. Tsiknakis, and P. Vorgia, "Focal epileptic seizures anticipation based on patterns of heart rate variability parameters," *Comput. Methods Programs Biomed.*, vol. 178, pp. 123–133, 2019.

[43] A. Popov, O. Panichev, Y. Karplyuk, Y. Smirnov, S. Zaunseder, and V. Kharytonov, "Heart beat-to-beat intervals classification for epileptic seizure prediction," in *2017 Signal Processing Symposium (SPSympo)*, Sep. 2017, pp. 1–4.